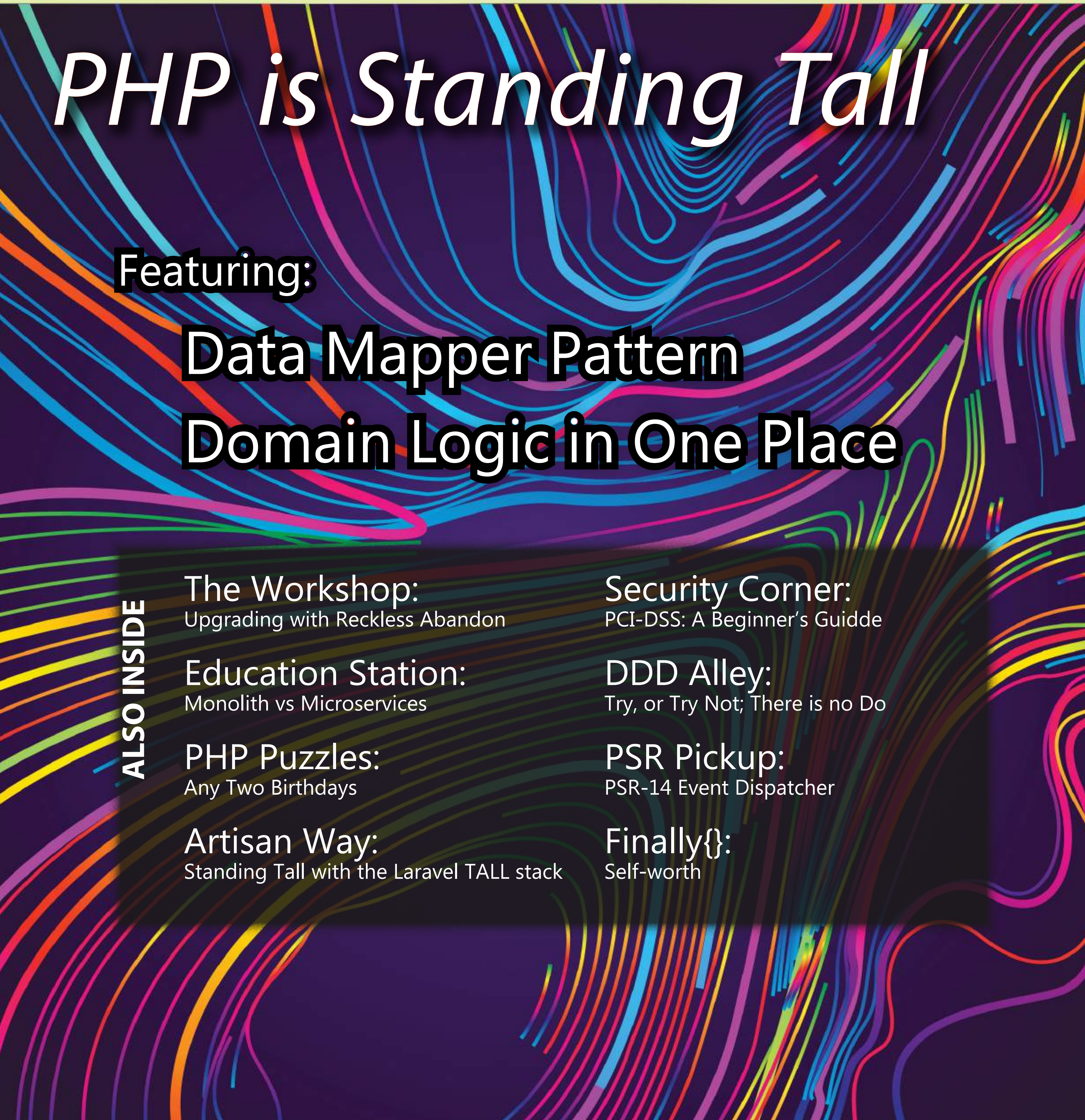
www.phparch.com

January 2023
Volume 22 - Issue 1

php[architect]

The Magazine For PHP Professionals



PHP is Standing Tall

Featuring:

Data Mapper Pattern
Domain Logic in One Place

ALSO INSIDE

CONTENTS

JANUARY 2023
Volume 22 - Issue 01

Edited in a state of celebration

php[**architect**] is published twelve times a year by:

PHP Architect, LLC
9245 Twin Trails Dr #720503
San Diego, CA 92129, USA

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!

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

Print ISSN 1709-7169
Digital ISSN 2375-3544

Although all possible care has been placed in assuring the accuracy of the contents of this magazine, including all associated source code, listings and figures, the publisher assumes no responsibilities with regards of use of the information contained herein or in all associated material.

The Year of php[tek]

John Congdon

Happy New Year from everyone here at PHP Architect. We hope that you are ready to make 2023 your best year yet, and we want to be a HUGE part of your journey.

We saw many changes internally at PHP Architect last year. Change is hard to deal with at times, but we are getting through those changes to be a better overall company. But not all change is bad, and in fact one of the biggest changes is that we had decided to bring our beloved php[tek] conference back this year. So hold on over the next few months as I'm sure it will be a huge focus of our magazine.

Since last month, the speakers and schedule have been posted. These two pieces of information are often a driving factor for people to purchase tickets. I'd like to pull your attention to just a few of the amazing presentations lined up. Derick Rethans, creator of XDebug, will be giving a couple of amazing presentations on PHP Internals and Debugging. If you have not seen Keith "Danger" Casey give a presentation, then you are in for a great time, he has a way of making presentations entertaining and informative. Nuno Maduro, creator of the Pest testing framework, will be giving an introduction to Pest and also Rust For PHP Developers.

These are just a few of the amazing presentations you'll find in Chicago, May 16-18, 2023. Learn more at https://tek.phparch.com

Want to join, but don't think you can convince your boss or manager to let you go? We have an email template that you could try. If that still doesn't get the job done, email us and let us do your dirty work for you.

Now for this month's magazine. Alexandros Gougousis will shine a light on database patterns with "The Data Mapper Pattern". Stathis Geogiou will help tighten up our business rules by putting our "Domain Logic in One Place".

Our columnists continue to contribute great material as well. Chris Tankersley will have you thinking about the architecture of your application in Education Stations with "Monolith vs Microservices". Is 2023 the year you make the switch from one to the other?

Oscar Merida brings us a puzzle to have us working with dreaded dates and times in this month's PHP Puzzles column, "Any Two Birthdays". Remember that while these puzzles seem easy on the surface, they are designed to give you a semi-simple problem to practice with. You can use these puzzles to try new design patterns, or maybe new technology. Want to learn TDD? Practice using a puzzle before trying to incorporate it into a mission critical application where you may be overwhelmed.

Over in Security Corner, Eric Mann wants to get you started in your understanding of credit card compliance with "PCI-SS: A Beginner's Guide". Frank Wallen shows us the "PSR-14: Event Dispatcher" that we should follow to help simplify our code. Joe Ferguson wants to take you into The Workshop and show you how to live dangerously in "Upgrading with Reckless Abandon Part One". Edward Barnard takes us own DDD Alley and brings us "Try, or Try Not; There is No Do" where he wants to extend the transactional boundary concept around RESTful API requests and responses.

We bring a new columnist, Matt Lantz, in for Artisan Way where he shares "Standing Tall with the Laravel TALL Stack".

And finally, Beth Tucker Long wants you to think about your "Self-worth". Mental health is so important for everyone, and we here at PHP Architect hope that everyone reading this has the resources and people around them to achieve their best mental state.

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.

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

Download the Code Archive:

https://phpa.me/January2023_code

Data Mapper Pattern

Alexandros Gougousis

ORMs are great for Rapid Application Development. You can get started very quickly. The price you pay is the vast complexity they hide under the hood. As your system grows, that complexity can bite you if you are not careful enough. Moreover, ORM packages create heavy objects that can make your life hard when doing batch processing. Knowing the basic principles and challenges when mapping an object to a relational database is a great blessing. This knowledge will help you make some important decisions as your application grows.

As you may know, there are two popular approaches when it comes to persisting a domain object (or any object).

1. The Active Record pattern, followed by ORMs like Eloquent and Propel
2. The Data Mapper pattern, followed by ORMs like Doctrine and Cycle

This article will explore the Data Mapper pattern.

The Object

Let's start by assuming that our domain contains a Product model, Listing 1, that needs persistence.

The Mapper

The Data Mapper approach moves the knowledge of persisting and loading entities to and from the database to a Mapper class. This class stands between your domain object and the database. The mapper class knows about the internal structure of your domain objects, the database schema, and how to move data between these two. Hence, your domain objects can even be unaware of the database's presence.

Listing 1.

```
class Product {
    private $id;

    private $title;

    private $brandName;

    private $catalogPrice;

    public function __construct(
        $id,
        $title,
        $brandName,
        $catalogPrice
    ) {
        $this->id = $id;
        $this->title = $title;
        $this->brandName = $brandName;
        $this->catalogPrice = $catalogPrice;
    }
}
```

Listing 2 is a mapper skeleton that contains an example of methods we could probably expect from our mapper.

The first problem we will face implementing these methods is that ProductMapper will need to access the properties of the Product object to update or delete the corresponding database row. There are several ways to do this. We can use reflection (or a similar technique) to bypass the language's visibility rules. We can place both the domain model and the data mapper in the same package and use some type of package-level visibility if the language supports it. Or, we can make the properties public (directly or through getters). All of them have their pros and cons. For the sake of this article, we will use a simple reflection wrapper as shown in Listing 3.

Now that we can access a Product's properties like this:

```
$publicProduct = new ReflectionWrapper($product);
$title = $publicProduct->get('title');
```

Listing 2.

```
class ProductMapper {
    public function create(
        $title,
        $brandName,
        $catalogPrice
    ): Product { }

    public function update(Product $product) { }
    public function delete(Product $product) { }
    public function findById($id): Product { }

    public function findByTitle(
        $title
    ): ProductCollection { }

    public function findExpensiveBrandedProducts(
        $costLimit
    ): ProductCollection { }

    public function updateOldCollectionsByPercent(
        int $percent
    ) { }
}
```

Listing 3.

```
class ReflectionWrapper
{
  private $object;
  private $reflectedObj;

  public function __construct($object)
  {
    $this->object = $object;
    $this->reflectedObj = new ReflectionClass($object);
  }

  public function get($propertyName)
  {
    $property = $this->reflectedObj
                     ->getProperty($propertyName);

    if (
        $property->isPrivate()
        || $property->isProtected()
    ) {
      $property->setAccessible(true);
      $value = $property->getValue($this->object);
      $property->setAccessible(false);
    } else {
      $value = $property->getValue($this->object);
    }

    return $value;
  }
}
```

Let's start the data mapper coding with the `create()` function. See Listing 4.

Listing 5 shows the update and delete functions that will make use of the ReflectionWrapper:

Assuming we can allow objects to be without an ID until they are persisted, we can let the database handle the generation (using an auto-increment column) at the time of persistence and use a constructor without an ID. Doing so is very common in ORMs, though it raises other issues, for example:

- the uniqueness of IDs in a clustered database
- the coupling between the domain object and the database technology
- how are we going to access the ID to set it, etc.

The next step is to see what is going on with the methods that load data from a database. First of all, we need to put the code (Listing 6.) that converts one or more database rows (in the form of an array) into a Product object. The code will be common for all finder methods.

We can now start writing our finder methods as shown in Listing 7.

Some people may be troubled here. The application code is calling the method `findByTitle()`. How does the application code, which calls this method, know that there is a title

Listing 4.

```
class ProductMapper
{
    private PDO $pdo;

    public function __construct()
    {
        $this->pdo = // fetch the PDO object
    }

    public function create(
            $title,
            $brandName,
            $catalogPrice
    ): Product {
        $sql = 'INSERT INTO products
                (title, brand_name, catalog_price)
                VALUES (?, ? ,?)';

        $pdoStatement = $this->pdo->prepare($sql);
        $pdoStatement->bindValue(
                1,
                $title,
                PDO::PARAM_STR
        );
        $pdoStatement->bindValue(
                2,
                $brandName,
                PDO::PARAM_STR
        );
        $pdoStatement->bindValue(
                3,
                $catalogPrice,
                PDO::PARAM_INT
        );
        try {
            $this->pdo->beginTransaction();
            $pdoStatement->execute();
            $productId = $this->pdo->lastInsertId();
            $this->pdo->commit();
        } catch (PDOExecption $e) {
            $this->pdo->rollback();
            throw $e;
        }

        $product = new Product(
           $productId,
           $title,
           $brandName,
           $catalogPrice
        );
        return $product;
    }
          ...
}
```

Listing 5.

```php
class ProductMapper {
        ...

  public function update(Product $product) {
    $publicProduct = new ReflectionWrapper($product);

    $sql = 'UPDATE products SET title = :title,
        brand_name = :brandName,
        catalog_price = :catalogPrice WHERE id = :id';

    $pdoStatement = $this->pdo->prepare($sql);
    $pdoStatement->bindValue(':id',
          $publicProduct->get('id'),
          PDO::PARAM_INT);
    $pdoStatement->bindValue(':title',
          $publicProduct->get('title'),
          PDO::PARAM_STR);
    $pdoStatement->bindValue(':brandName',
          $publicProduct->get('brandName'),
          PDO::PARAM_STR);
    $pdoStatement->bindValue(':catalogPrice',
          $publicProduct->get('catalogPrice'),
          PDO::PARAM_INT);
    $pdoStatement->execute();
  }

  public function delete(Product $product) {
    $publicProduct = new ReflectionWrapper($product);

    $sql = 'DELETE FROM products WHERE id = :id';

    $pdoStatement = $this->pdo->prepare($sql);
    $pdoStatement->bindValue(':id',
          $publicProduct->get('id'),
          PDO::PARAM_INT);
    $pdoStatement->execute();
  }

     ...
}
```

Listing 6.

```php
class ProductMapper {
     ...
    protected function convertRowsToProducts(
        array $rows
    ): ProductCollection {
        $products = new ProductCollection();

        foreach($rows as $row) {
            $product = $this->convertRowToObject($row);
            $products->add($product);
        }

        return $products;
    }

    private function convertRowToObject(
        array $row
    ): Product {
        $product = new Product(
            $row['id'],
            $row['title'],
            $row['brand_name'],
            $row['catalog_price']
        );

        return $product;
    }

    ...
}
```

column in the database schema? Does the schema information leak into the application layer? No! The application layer does not know about the schema! The title is a property of the Product model of your business domain. We don't care whether this title property is persisted as a "title" column or the table where it is persisted be named "product". It could be saved with another name or even inside a greater data structure that would, maybe, be saved in JSON format. Listing 8 shows another finder method.

Where does the definition of an expensive branded product lie? It is part of the domain logic, of course! Which branded products are considered expensive is expected to be something that will come from the domain experts. Hard-coding it inside the mapper would be considered a bad practice. A better approach would be to make it a constant in the Product class or, even better, a setting set through some admin interface. Another approach would be to remove this knowledge from the mapper. We could refactor the method to something like:

```php
findBrandedProducts($brandName, $minPrice = null)
```

Now, the semantics of what we are looking for have been moved one layer up, to the caller of the mapper. If we have multiple cases with different context for `$minPrice`, then this method would be a better choice.

There is an important and very common situation we need to deal with moving forward. Imagine a product listing page that provides us with many filtering options (probably, as many as the product properties). We can't define one method for every filtering combination. A common approach to this is to use abstracted finder methods. If you have used Doctrine, calls like this may look familiar to you:

```php
$productRepository->findBy([
    'brandName' => 'Adidas',
    'title' => 'Ultraboost 21'
]);
```

Listing 7.

```
class ProductMapper {
  ...
  public function findById($id): Product {
    $sql = 'SELECT * FROM products WHERE id = :id';

    $pdoStatement = $this->pdo->prepare($sql);
    $pdoStatement->bindValue(
      ':id',
      $id,
      PDO::PARAM_INT
    );
    $pdoStatement->execute();

    $row = $pdoStatement->fetch(PDO::FETCH_ASSOC);
    $product = $this->convertRowToObject($row);

    return $product;
  }

  public function findByTitle(
    $title
  ): ProductCollection {
    $sql = 'SELECT * FROM products ' .
           'WHERE title = :title';

    $pdoStatement = $this->pdo->prepare($sql);
    $pdoStatement->bindValue(
      ':title',
      $title,
      PDO::PARAM_STR
    );
    $pdoStatement->execute();

    $rows = $pdoStatement->fetchAll(PDO::FETCH_ASSOC);
    $products = $this->convertRowsToProducts($rows);

    return $products;
  }
...
}
```

Again, you may think that schema knowledge is leaking to the application layer. Yes and No. Theoretically, “brandName” and “title” are concepts of our business domain. I am saying concepts and not properties because the fact that we are passing them in the constructor doesn’t necessarily mean that they are internally implemented as properties. Actually, we shouldn’t care if they are object properties as long as the caller understands their meaning and can use them properly when calling the finder method. Another way to implement this is a finder method with a signature like that:

```
public function find(
    ?string $title = null,
    ?string $brandName = null,
    ?int $price = null
)
```

Listing 8.

```
public function findExpensiveBrandedProducts(
       $brandName
): ProductCollection {
    $sql = 'SELECT * FROM products
            WHERE brand_name = :brandName
            AND catalog_price >= :priceLimit';

    $pdoStatement = $this->pdo->prepare($sql);
    $pdoStatement->bindValue(
        ':brandName', $brandName, PDO::PARAM_STR
    );
    $pdoStatement->bindValue(
        ':priceLimit', 1000, PDO::PARAM_INT
    );
    $pdoStatement->execute();

    $rows = $pdoStatement->fetchAll(PDO::FETCH_ASSOC);
    $products = $this->convertRowsToProducts($rows);

    return $products;
}
```

That will build the query taking into account only the properties with a provided value. What happens when we have to search using inequalities? You could work with upper and lower boundaries:

```
public function find(
    ?string $title = null,
    ?string $brandName = null,
    ?int $minPrice = null,
    ?int $maxPrice = null
)
```

Though the method signature will soon become overly complex. Another way is to allow comparison operators as parameters:

```
$productRepository->findBy([
    ['brandName', '=', $brandName],
    ['catalogPrice', '>=', 1000]
]);
```

Which, again, increases the complexity of the implementation. A third one is to make a query builder available outside the mapper. A snippet of code using Doctrine’s query builder looks like: (See Listing 9)

I would resist the last one. It allows details about the schema and the persistence technology to leak into your business logic. It also creates a strong coupling between your application and the ORM package you use (if you use one). What is more, it lacks expressiveness. It doesn’t say a thing about what we are looking for and why.

Listing 9.

```
$q = $this->createQueryBuilder('p')
    ->where('p.brandName = :brandName')
    ->setParameter('brandName', $brandName)
    ->andWhere('p.catalogPrice >= :priceLimit')
    ->setParameter('priceLimit', 1000)
    ->getQuery();

$q->getResult();
```

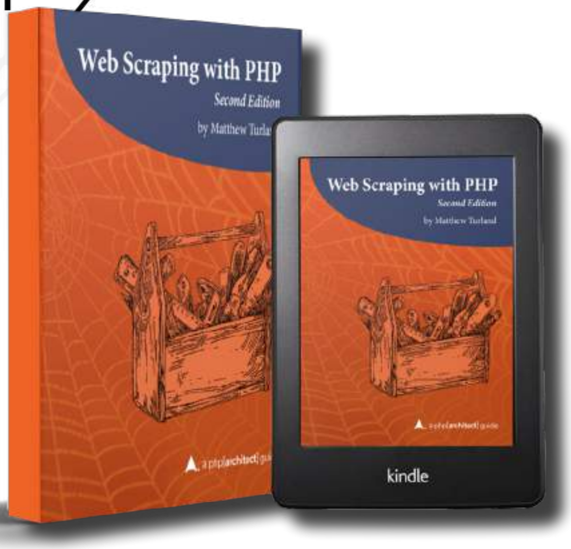
Closing Thoughts

Data mappers are destined to handle complex persistency cases. Here we have only scratched the surface. However, do not think that every feature implemented by a heavy ORM, like Doctrine, should be part of the data mapper used by your application. To give an example, you may not need a Unit Of Work. Many times, it may hinder more than help. An Identity Map may not be required for some mappers, especially if the objects returned by the mapper are immutable.

When a custom implementation of mappers is used, like in our example, having one mapper for each domain object is expected. You may be familiar with Metadata Mapping if you have used Doctrine ORM—the mapping information lies in metadata expressed, for example, as an annotation on your domain model class. If Metadata Mapping is used, it makes sense to have one abstracted mapper for all of your objects. Since data mappers are designated to handle more complicated cases of persistency, you can imagine the complexity of implementing such an abstracted mapper. There are also many choices between these two ends, for example, using one data mapper per aggregate or module.

Extensive use of data mappers from within the domain objects can be avoided by defining associations between domain objects and providing a method to lazy load their associated objects. By that, we can move from one domain object to another using a mechanism hidden in a common abstracted ancestor of all domain objects instead of explicitly using a data mapper. On the one hand, this removes complexity from your domain layer, but on the other hand, the complexity is not really gone but hides in some dark corners of the ancestor class of your models.

Alexandros Gougousis is a software engineer with an academic background in Electrical & Computer Engineering. He started as a system administrator, switched to programming in 2010 and since 2017 he is focusing exclusively on the back-end of web application development. Alexandros writes a developer blog. In his free time, he loves reading history and playing board games.



php[tek] 2023

Early Bird Tickets Now on Sale

Here are some of the speakers

Nuno Maduro

Joe Ferguson

Jana Sloane

Beth Tucker Long

Derick Rethans

Eric Mann

David Quon

Leslie Martinich

Jason McCreary

For our full speakers list, head over to tek.phparch.com/speakers/

May 16-18, 2023

Sheraton Suites Chicago O'Hare

tek.phparch.com

Brought to you by your friends at

Domain Logic in One Place

Stathis Georgiou

In this article, I want to examine one of the most common problems I have faced while working on many applications, legacy or not. I will also suggest an alternative solution, but first, I will need to lay out why I think this is happening and what we could look out for to avoid it in the first place. I am unaware of an official name for this phenomenon; however, I call it "scattered business logic". It simply means that the business rules which describe the problem and its solution often are spread out in multiple files and even technologies instead of being in one place.

Why It is Happening

As developers, we try to write code that describes and solves real-world problems. Every application we build tries to satisfy the needs of real people and, once more, solve their real-world problems.

As developers, we also love technology and making things work. These two characteristics often make developers focus more on the implementation of the solution and less on the problem which needs to be solved. This state of focus is affecting, of course, the development process from start to finish. It leads us to prioritize, for example, the usage of tools (just because it is fun to use them) instead of delivering something meaningful to our users.

All the top-tier developers I have worked with had a very direct approach to how to tackle a new feature required by the users. They were good problem solvers like most of us, but their focus was always on solving the users' problems, not on the technologies used to provide the solution. So, when they were faced with a problem to solve, they always first tried to avoid building a new feature to solve it. That was an eye-opening experience for me; the people who were the most skillful in the team tried to offer a solution for the users with the existing tools of the platform or even eliminate the problem by approaching it from different angles. At the same time, I was ready to dive into my editor and code out a solution.

I have often found myself many times trying to justify the use of a new data store that everyone is talking about (and is a cool and shiny toy) instead of working towards providing a solution that matters for the users of my app. At the end of the sprint, I always deliver the feature, being so deep in the code and its intricacies, I almost always forget why I was doing all the coding.

On top of all these challenges, we frequently receive the requirements of a new feature in so many distinct ways (tasks, tickets, slack messages). Making it hard in the first place to extract meaningful business rules, which need to be translated into clean, meaningful, and easy to work on code. Once more, the focus is key here; we can focus on just making the application "do" the things requested. Open our fancy editor, throw in some arrays and classes that transfer data around, save them in the store, and close the ticket. Next! This would be an easy and fast way to do everything. We satisfied our clients because the app is doing what they wanted, we did it fast enough because we jumped into the implementation straight away, and we used the document storage everyone is talking about. All boxes checked.

The problem with this approach will not show up until the next time you are called to add something on top of that feature, and even then, it will be too early for it to be big enough to scare you into refactoring it. It is only after a handful of iterations of the same feature that everything will start to be harder to do, requiring more time and brain power.

All the complexities mentioned above can and usually (in my experience) affect the development process and how we write and organize our code. We can easily get lost in the different/conflicting interests that manifest themselves in every opportunity, resulting in turning all of our efforts and focus on the wrong, not-so-important things.

Problem: How the Misguided Focus Affects Our Code

If we don't address all the issues mentioned above, and we are not conscious that we need to face and tackle the difficulties presented by them as soon as possible, we will probably fall into one of many pitfalls. Writing code before planning what you will build is one of these pitfalls. We need (at least) to write down what we want to achieve in every code session and how it will affect our users. These pre-dev notes can be taken just in a notepad, and they will still be enough to stir our focus in the right direction. On the other hand, jumping immediately to implementation can lead us to focus more and more on how the code comes together and how we can make a fancy query to save all the info we want, instead of focusing on the actual info (business logic) and how it is used to solve a real-life problem. Consequently, we scatter important information across multiple files as we just focus on getting the job done fast, and it almost feels like we hack something together, and everything works out of luck.

In the example below, I will try to showcase the things that can go wrong, and although the example will be very simple, it can be enough to make visible the outlying dangers and the uncertainty of the code. (See Listing 1)

Listing 1.

```php
<?php
// $inputs represents the inputs of the request

$book['title'] = $inputs['title'];

if (isset($inputs['discount'])) {
   $book['discount'] = $inputs['discount'];
}
$this->repository->add($book);
```

Let's take a look at the simple example in Listing 1. There is a variable called inputs that holds the request's payload and, more specifically, the information for a new book inserted by a user. We have the title and optionally the discount on the new book. At the end of the script, the book is passed to a repository function to save it in a data store.

When I see something like the example in Listing 1, I always focus on the fact of how many things can just be implied in a small number of lines of code. For example, what is the default discount for a book? Where is it defined, is it defined in the add method of the repository, or maybe it is defined as a column default in the database? What happens if the title is just one letter? Is that acceptable? Are these rules enforced in all the places where our application creates a new book?

Another quality missing from the example code above is the declaration of the code's intention. In a simple example like this, you can "imagine" what the code is doing, but the developer who wrote it put zero effort into making the intention visible to the reader. There are many resources for intention-focused code but a simple rule I follow is to imagine being a first-time reader of my code and asking myself "what is the intention behind all that?". If the answer is not provided from the names of the variables and functions used, I try to rewrite the code until it does.

Every time I had to change something seemingly simple, I found myself searching in a gazillion files with zero correlation between them. Sometimes, I had to go as far as the table defaults to realize why the system was working as it did. There was no single place to define what a book is and how it behaves in the application. Back then, I felt things shouldn't be so hard, but I did not have a solution. Convoluted logic scattered across multiple files and many times repeated, again and again, is what I saw in almost any legacy application that I have worked on. As a beginner at the time, I always thought this was the only way to go, but it doesn't have to be, and maybe there is also a good solution to start with.

Suggestion: Use Your Objects

If we take a step back and try to imagine a better solution, simple enough so everyone can pick it up when they work on our code, we may turn our heads on the OOP side of the language we use. We can just use the raw PHP materials is offering to us. These can be our classes and their objects. Imagine instead of an array transferring data around as in the example above; you had an object—not any object, but a "book object" that has properties like `discount` and methods like `applyDiscount` that contain any logic relevant to a book. OOP is giving us all the tools to translate the real world in our classes and their instances. No need for fancy, advanced techniques or patterns here. We can create simple classes that just describe the information that solves the problem and the actions that can be done upon this information. These classes are usually called entities, as they describe a real-world entity in our application. Even better, you can name these classes and their methods with the real-world words you use to describe the actions that the application performs, so you can reveal the intention of your code. If the user "can set a book title" and "apply a discount" to it, there you go; the user case helped you tackle one of the harder things to do in programming: name your variables!

In a hypothetical question from a business person to a developer, "What is the default value of a book discount in our system", the developer should have one and only one place to look to get the answer. (See Listing 2 on the next page)

In the code example in Listing 2, we have a class called Book. It has two properties that hold information for the title and the discount and two methods to set these properties. Each method is called in the constructor instead of directly assigning values to the properties, so the rules defined for each property are always validated.

By just peeking in the Book class from Listing 2, we can extract that the title is a string, and it cannot be less than three characters. We also see the discount defined as a float which cannot be a negative number and has the default value set to 0.

The name of the methods is also a very useful tool, as mentioned before. The methods are used to hold the logic and the business rules but also to describe what actions a book can perform by naming them with terms used by the application's users. The next developer looking into this class will be able to understand what a book is in the system and what actions can do.

With just one class and basic OOP knowledge, we have a good structure of a book defined in our application that describes what a book means in the real world for our users. This class also allows us to define all the actions that can be performed in a book by a user and create a method for each action. The book class can be utilized across different scenarios, forcing us to always comply with the same rules without duplicating the same code and business logic.

Listing 2.

```
<?php

class Book {
   protected string $title;
   protected float $discount = 0;

   public function __construct($title, $discount = 0) {
      $this->setTitle($title);
      $this->applyDiscount($discount);
   }

   public function setTitle($title) {
      if(strlen($title) < 3) {
        $str = 'Title must be more that 2 characters';
        throw new Exception($str);
      }
      $this->title = $title;
   }

   public function applyDiscount($discountAmount) {
      if($discountAmount < 0) {
        $str = 'Discount cant be a negative number';
        throw new Exception($str);
      }
      $this->discount = $discountAmount;
   }
}

// In the class's client file

// $inputs represents the inputs of the request
$book = new Book($inputs['title'], 1);

$this->repository->add($book);
```

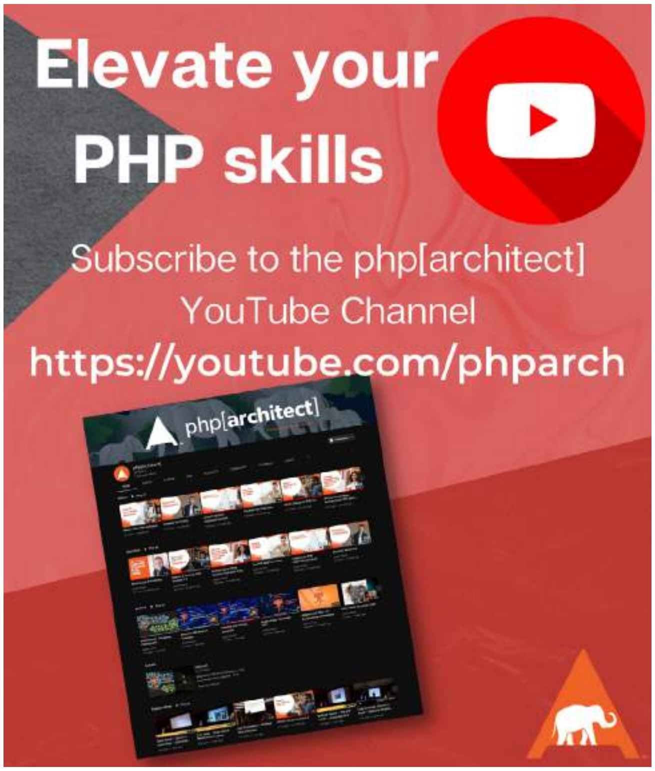
By using these simple classes, we can find a suitable place for our business logic to live in. All the logic can be inside the methods of the related entity. With the separation of concerts in mind, that entity class can just care about performing the actions on the data and not worry about anything else (e.g., how all the info is persisted, etc.). In a more convoluted business domain, you can use multiple object hierarchies to express the business logic, which is when design patterns are very useful.

Conclusion

Simple OOP knowledge can help us to describe the real-world domain our application lives in and write meaningful and intention-revealing code. There is a place to use everything, but my humble opinion is to try and use objects to describe entities (and their actions) of our system instead of data holder arrays. Also, we must be mindful that someone else will read our code and try to work on it. Six months later, that ‘someone else’ can be our future self, and we want to be helpful to them. We can try to keep all our business rules in one place, in the entities, and not scatter it everywhere because it is the fastest way to solve the problem. We need to put some thought into every decision we make with the right goal in mind, which is to solve our user’s problems.

After reading this article, something else to chew on is that our job as developers is to solve problems for real people and not to write fancy code using cool technologies. We need to remind ourselves of our intentions (solve the users’ problem) all the time. We, either way, love to solve problems; this is why we chose this profession—we just need to focus on the right ones. There will be opportunities to solve a problem without writing a single line of code; grab these opportunities and be proud of not opening your editor to get the task done! To constantly remind myself, I have a post-it note on my screen that reads, “what does the business need?” to always remind me that I am providing a solution for the business, and of course, by extension, for the application’s users.

Stathis Georgiou currently lives in London, with his two kids and wife. He is a happy dad and a passionate developer who is always looking to improve himself and help others do the same. His favorite language to work with is PHP, which was love at first sight for him! He has worked with PHP for the last six years on various projects, small to very complex, and he is always amazed by how differently people use this language to solve their problems.



Monolith vs Microservices

Chris Tankersley

Which should you use…Monolith or Microservices? Neither is better or worse, so let’s look at them more closely to figure out which will work for you.

> *Developing software is hard. Developers spend a lot of time trying to come up with ways to make development easier, especially on projects that will be long-lived. We have all run into that app that we are afraid to update or refactor for fear of unknown breaks. The code has been working fine for years; why poke that bear?*

This has given rise over the last few years to the idea of microservices and how they help solve the problem of monolithic codebases. As with everything in technology, this has spawned zealots on both sides of the argument, with no real side being able to “win.” We are locked in an eternal battle while being caught in the middle. If only the answer were as easy as the question of, “vim or emacs?”. (The answer is “vim”).

Grab your sword and shield, and let’s wade into the flame war.

What is a Monolith?

Monoliths in the real world are large, upright structures, generally made out of stone. Think of the rocks used in Stonehenge, or in extreme cases, huge rock structures like Uluru in Northern Territory, Australia. They are ancient structures that have withstood the test of time and continue to exist long after everything else around them has eroded away.

A Monolithic codebase is pretty much like its physical counterpart in the real world—a singular, large, giant object that, in many cases, few people understand due to a lack of tribal knowledge. The entire codebase is functionally one entity, even if using proper techniques like Domain Driven Design or solid object-oriented programming. You cannot find components that can be pulled out or replaced, and much of the code is not standalone.

A good example is WordPress. There tends to be little inside the core WordPress system that you can easily pull out and use in other applications. Parts like the post system may be interactable with plugins, but you would have a hard time pulling out just the architecture of working with posts to use in your own application. It has nothing to do with any failure in WordPress; it is just that monolithic codebases are much more intertwined.

That is not to say monolithic codebases are a bad thing, or any one design choice causes this. When developers sit down to solve a problem, we tend to look at things from a top-down perspective. Solutions to problems start at the macro level (let’s build an easy-to-use blogging system), and the micro problems become intertwined. Posts need an author, so we’ll just join that table in the SQL query that gets the post. Problem solved!

The fact of the matter is that most codebases in the world are monolithic codebases. Let me stress that that is not a problem. If I am being honest, the longest-running systems I have come across in my nearly twenty years of programming have been monoliths, and many of them continue to function just fine. The problem is not the code.

The world of computing is very quick to find faults in our designs and iterate on them. Many of the “hot technologies” we work with today were actually invented years ago. Software-as-a-Service platforms? We had server-client platforms all the way back in the 90s. Server-side rendering? That’s been PHP’s schtick forever. Returning HTML in a JavaScript callback? Drupal 6 waves from its grave.

Monolithic architectures have endured, not because we did not have anything better, but because they tend to be incredibly resilient.

Why Do People Hate Monoliths?

People hate monolithic codebases for a few reasons. Some of them are technical, but a few are organizational. They are all solvable problems.

One main reason developers hate monoliths is that they are almost always a black box. The lack of tribal knowledge around the why disappears into the binary ether as the original developers move on to other projects and jobs. The “why” is vastly overshadowed by the “how”, and the codebase turns into hieroglyphics that no one knows how to read.

Questions like “How does this turn an XML response from the API into a value object?”, “Why is this route the one that’s selected?”, or “I see that this job is getting fired, but how in the world does it get its parameters?” become the most pressing issue when a new developer is fixing a bug. They do not have time or even a person to ask, “Why does the system function this way?” The original developer could have thought they were leaving a beautiful, elegant, and easy-to-understand system for people to maintain. Still, it’s all just gibberish at the end of the day.

I will give a personal example. I spent two years at Vonage, maintaining their PHP SDK. I took it over from Tim Lytle, and I quickly had to turn to him and ask, “What in the world were you thinking?!” While a few of the answers were 2 am in a hotel room at a conference was a bad time to be building an SDK, in many cases he had valid arguments and reasons for the code he wrote. His explanation made me understand the "why" inside the SDK.

So I rewrote it. I deprecated a bunch of weird, but now fully understood, edge cases. I cleaned up a bunch of the tests and rearchitected things in a way that I thought decoupled things properly while still letting end users get what they needed. I was, and still am, proud of what I wrote.

I left for about a year and came back. Jim Seconde had taken my place at Vonage. I have been back for almost eight months at the time of this article, and Jim still has questions about why I did things. The things that I thought were clear, it turns out, were obtuse. The edge cases that I had to handle were apparent to me but not to Jim. This is not a fault of Jim—this is a fault of my knowledge being lost, and where I did leave nuggets of info, I was not clear enough.

The next issue is the perceived brittleness of monolithic systems. Many of them were developed when things like test-driven development were not considered best practices by the vast majority of developers. Test Driven Development is not a new idea, having been around since 1999, but it was not a widely used paradigm. When you are tasked with altering a system of tightly interconnected systems, many developers get lost in the rat's nest of code.

The lacking tribal knowledge of the "why" of the interconnected systems turns into lacking tribal knowledge of the "how" systems interplay. Changing how the user system works impacts how the Posts system works, and our posts stop showing up on the site. Why? We have no idea because there are no tests to tell us exactly where the code is breaking.

So we begin to try and add tests. Our tests are now thirty lines of mocks that we hope are correct to test a single method call. What conceptually should have been a few hours of work has ballooned into a week's worth of just detective work, and we are no closer to having the user's e-mail address displayed on a post than we were before. A weird mixture of imposter syndrome sets in ("why can't I understand how this works?") and along with a healthy dose of Dunning-Kruger ("that previous dev was an idiot to do it this way, I need to rewrite this") you schedule a sprint to just redo all the code. (Spoiler: three sprints later, it's still not rewritten, and the original system now resides next to the "new" system).

The final reason tends to be organizational. As a system grows in size, there is a greater potential for more people to help maintain the system. This causes friction when one team ends up butting into another team due to the interconnected systems. I do not remember where I heard the phrase, but "the slowest part of development is dealing with other teams" seems to be a universal idea.

Your work is now dependent upon another team's work and schedule, and their roadmap may be different from yours. Work stalls as you try and work with the team leads and managers to get into their roadmap, and you have to now deal with this branch of code that is unfinished because something else needs updating that you cannot easily do.

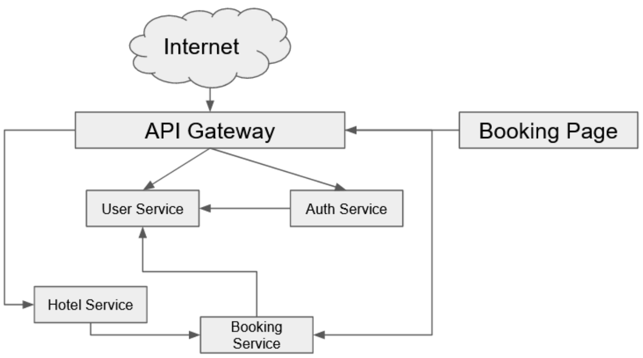
What are Microservices?

Tiny monoliths.

Microservices are an architectural idea that breaks down a singular application into various problem domains. Each domain is maintained as a small application by a small team that can act independently of others. These applications are then wired together, many times via APIs, to solve the overall macro problem being presented. Each team is able to work and deploy code independently of the other teams.

The keen among you will notice that the previous paragraph spends a lot of time talking about teams and problems and very little about technical issues. That is because microservices should be used to solve that final organizational problem that monoliths have where teams block other teams.

Figure 1.



When a project decides to use a microservice, we tend to see diagrams like those in Figure 1. We see various services that are deployed, independent applications that tend to solve individual problems. Need to handle authentication? Build an authentication service that just deals with matching a user to a password. Now we need a way to look up users, which turns into another service. That is OK, though, because our Booking service also needs to look up users.

Each application may break down in a few ways, but the overall idea is that each problem domain (the booking service, the auth service, etc.) can be handled by teams specializing in that system. Anyone needing to depend on those services can talk to the service rather than couple themselves to specific blocks of code inside a monolithic structure.

A proper microservice setup means that the individual services can update independently. If we need to add additional options for searching for hotels, we can update the Hotel Service with the new features and not touch the other services at all. They can later upgrade to the newer features when it makes sense for them. The hotel team does not need to worry about breaking the booking system. Conversely, the hotel service should not worry about the authentication system updates breaking anything in the hotel service.

Microservices live and die by their ability to abide by the contracts that each service exposes. Each service strives not to break those contracts. This is both a technical requirement as well as an organizational requirement.

You may want to rethink using microservices if you have a single engineering team. As I will bring up many times, microservices tend to solve an organizational problem much more than they solve a technical one because they help avoid blocks between teams. If you have a single engineering team, there is no one to block them but themselves.

There is also the fallacy that to scale, you will need microservices. For almost every application out there, this is fundamentally false. Monoliths scale just fine. In fact, monoliths have scaled for tens of years both horizontally (more servers) and vertically (larger servers). Most developers will not have scaling issues like Alphabet or Meta that necessitate some services being able to scale independently of others.

Treat each microservice like its own monolithic service. Good practices like test-driven design, clean coding, and adherence to standards will still be a fundamental need when designing microservices. If you skip out on writing good code…

Most Microservices are Just Distributed Monoliths

In theory, microservices sound like a good idea. What tends to end up happening is that the time and effort it takes to build a good microservice architecture can be quite a bit, so shortcuts are made. In a system with multiple teams, the contract between the systems often gets broken, forcing services to be deployed together as they get more coupled.

A good example is that many microservices will use REST APIs internally to communicate, but teams rarely decide on a versioning mechanism. So v1 of the authentication service goes out, and the front-end as well as some other back-end services begin to rely on it. A few months later, there is a decision to change how the tokens are generated, so v2 of the authentication system is written and deployed.

Many times these version numbers are just tags in a repository and have no basis in code. So when v2 of the authentication system is deployed, either everything that relied on the old v1 code will break or have to be upgraded and be available to release at the same time v2 of the authentication system is released. Backward compatibility is not thought of because why do you need it? The auth system is an internal system. Oh, and now it's too hard to support both because v2 is fundamentally incompatible with v1.

When your microservices become strongly coupled to other services, all you have done is break up the monolith into more monoliths. The overall application is no longer the sum of its independent parts—you are just deploying to more servers.

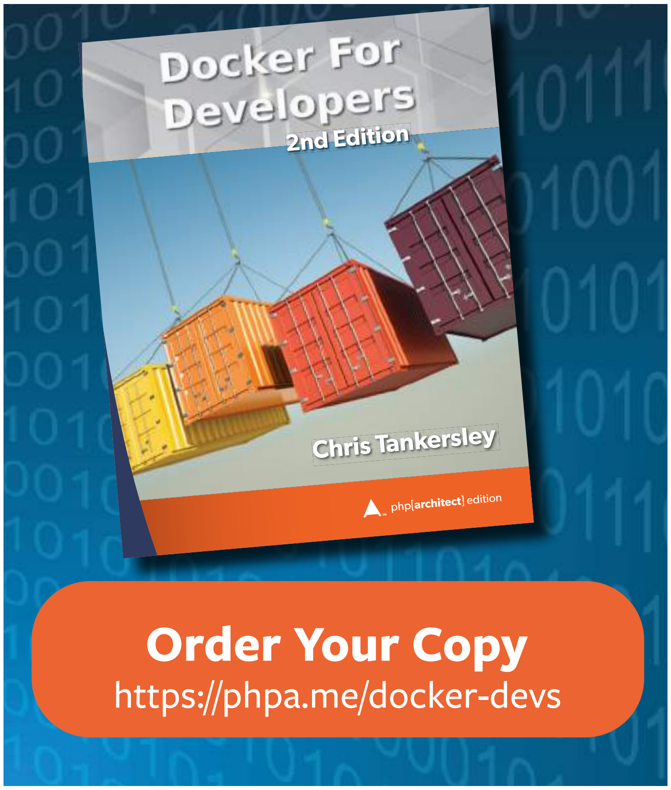
You can also run into many of the same issues with microservices as you do monoliths. As team members move on, the only saving grace is that the services should be small and hopefully easier to understand. That can be a double-edged sword. If a service becomes too complex, there is a good chance it will go through the same archaeological process a monolith does and be rewritten.

Which Should You Use?

What architecture should you use? You should use whichever makes the most sense for your application and organization. Neither is better or worse than the other—both can scale and be maintainable. One will make more sense for your team/teams and how you are organizationally aligned.

Next month we will take a deep dive into actually building a microservice-powered application so you can see exactly what goes into building one and the specific technical and architectural questions that can arise. It is much more than "that sounds like a service, make a new repo!"

Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of Docker for Developers and works with companies and developers for integrating containers into their workflows. @dragonmantank



PCI-DSS: A Beginners Guide

Eric Mann

Every developer should strive to not only build a quality application but also to ensure that security is baked in at every phase of development. Applications handling customer *payment* information are even more critical to secure. Firstly, it's just the right thing to do to ensure that you handle customer payment data appropriately. But if you want to work with credit cards, you're explicitly *required* to follow a set of standardized guidelines: PCI-DSS.

What is Pci-dss?

The major credit card providers—Visa, Mastercard, American Express, Discover, and JCB—gathered together in 2004 to create a standards body called the Payment Card Industry Security Standards Council (PCI SSC). This group worked to align the individual data security policies from each of the member organizations to create a single standard[1]: the Payment Card Industry Data Security Standard, or PCI-DSS[2] (most in the industry simply refer to this as "PCI").

PCI is the standard established minimum baselines that any enterprise must meet in order to work with payment processors while handling customer cardholder data. In other words, if you accept credit cards as payment for goods or services, you need to understand the compliance requirements behind PCI.

There is no legal requirement to comply with the PCI standard in the United States or abroad. However, failure to comply with the standard could result in the member payment processors refusing to work with your business. In addition, the standards have been incorporated into law by various municipalities—enterprises in Washington state are shielded from liability[3] in the event of a data breach if they were documented to be in full compliance prior to the incident.

There are four levels of PCI compliance[4], each determined by the number of card transactions processed in any given year by the business:

- Level 1: Merchants processing more than 6 million transactions per year
- Level 2: Merchants processing between 1 to 6 million transactions per year
- Level 3: Merchants processing between 20,000 to 1 million transactions per year
- Level 4: Merchants processing fewer than 20,000 transactions per year

The compliance and reporting requirements get progressively stricter as an enterprise graduates between the levels. When a small business starts, it's appropriate for them to self-attest compliance by completing a comprehensive questionnaire. By level 2, businesses must also submit a report on compliance to the banking institutions with which they work. Level 1 additionally requires an external audit to be conducted by a Qualified Security Assessor (QSA), who directly assesses your business practices and controls, then completes the report on compliance on your behalf.

Regardless of your enterprise's level of compliance, the goal isn't so much strict regulation (there are no laws *requiring* your compliance with PCI). Instead, the goal is protecting your business's reputation with customers as a reliable merchant and ensuring your right to collect payments in the eyes of the payment processors. Failing to remain compliant gives your competition an advantage and places your customers' data at risk.

Understanding Compliance As a Developer

It's often unnecessary for a developer to fully understand the intricacies of the PCI standard. Instead, it's a matter of asking some key questions while developing a system:

- How are customers making a purchase?
- When a customer submits their credit card information, where does it go?
- How are we protecting the flow and storage of data related to payment transactions?

Any data submitted to your application should be safeguarded by the system and the developers who maintain it. But *payment* information will always be held to a higher level of scrutiny. Who has access, through which systems, for what reasons … these questions will inevitably be asked by business partners, customers, or auditors.

The easiest way, as a developer, around these questions is to avoid handling the data in the first place!

External systems like Stripe can remove the burden of accepting and processing PCI data entirely from your application. Instead of credit card data

[1] https://phpa.me/pcissc-overview

[2] https://phpa.me/pcdsecuritystandards

[3] https://phpa.me/infolawgroup

[4] https://phpa.me/pci-compliance

flowing into your server, it goes to Stripe's server instead[5], and you receive an opaque token representing the card. Your application can leverage Stripe's API to act against that token directly—for charges, subscriptions, refunds, etc.—leaving your system entirely free of anything resembling PCI data.

Stripe also offers additional services for those who need to store PCI data in any way to ensure they are fully in compliance with the standards. They also work directly with QSA organizations[6] to aid in audits if and when they become necessary.

Similarly, platforms like Magento[7] help you integrate your application directly with secure payment gateways to help ensure compliance from day one. Regardless of which partner you choose, merely having a partner like Magento or Stripe drastically simplifies your job as a developer and lets you focus on the application instead.

A Living Standard

Like any other standard, PCI-DSS has evolved over the years. The latest version, 4.0, was released in the middle of last year[8] and is fully supported for organizations attesting to compliance today. For the entirety of this year, you can reasonably use either the latest version or its immediate predecessor, v3.2.1—in fact, many automated tools are still being updated to the newer version of standards during this transition period.

By the beginning of 2024, though, v3.2.1 is slated for retirement, and you *must* use the new standards for attestation.

The key updates in the new version of the standards include[9]:

- An explicit focus on network security controls rather than legacy firewalls and routers
- Core requirements around secure configurations beyond manufacturer defaults
- A renewed focus on *account* data and its protection
- New requirements about strong cryptography and anti-malware systems
- Standardization around authentication (passwords and MFA)

The updates are deep enough to the standards that anyone attempting to attest to their organization's compliance would be well incentivized to work with a qualified partner to ensure nothing is missed in the update.

Final Thoughts

In nearly every situation, the easiest path to achieving PCI compliance is not to handle customer payment data in the first place. You'll still need to ensure your application, systems, and network are well-structured, secure, and behave. But by removing payment information from the system, you have *drastically* simplified the scope of your job. If possible, work with a partner like Stripe to let *them* shoulder the risk of data management and the burden of compliance.

If an API integration partner isn't a fit, perhaps leverage a pre-packaged eCommerce solution like Magento. They'll handle the payment gateway integration for you and, again, shift the burden of PCI compliance off your team's shoulders.

So long as your business is collecting payment in exchange for its goods and services, PCI will be a critical conversation topic for your security and development teams. It's critical you still work to build secure software, monitor your application, and protect your customers' data. Working with a PCI compliant partner is not a magic pill to complete security, but it will absolutely simplify the path you take to get there.

Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: @EricMann

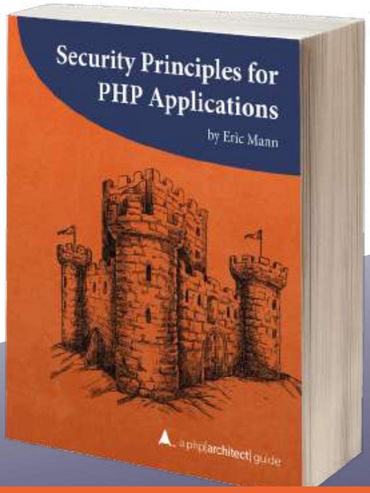
[5] *https://stripe.com/payments/elements*

[6] *https://phpa.me/stripe-compliance*

[7] *https://phpa.me/business-magento*

[8] *https://phpa.me/pci-security-standards*

[9] *https://phpa.me/pci-whatsnew*



Upgrading with Reckless Abandon

Joe Ferguson

This month we'll dive into upgrading a PHP 7.1 application originally built with the Laravel framework version 5.6 with reckless abandon. The *only* way this kind of upgrade can succeed is with a robust suite of tests. Tests allow us to leap major versions of our framework and dependencies and even PHP versions in some cases. Without a strong test suite, we couldn't possibly have the confidence to jump versions without thoroughly testing each change manually. If you'd like to jump versions for your projects, ensure you start by writing as many tests as it takes to provide confidence in major structural changes.

Laravel 5.6 was released on February 7, 2018, and stopped receiving bug fixes around August 7, 2018, and stopped receiving security updates on February 7, 2019. This application was built on PHP 7.1, and our goal is to move it to PHP 8.1 and the latest Laravel version, which is 9 at the time of this writing. Some big changes in 5.6 were dynamic rate limiting, Eloquent date casting, and Argon2 password hashing (for PHP 7.2+). We have several versions to jump to get to the current Laravel version. Before we worry too much about what *could* affect us, we should describe our application and how we got to running a Laravel 5.6 app on PHP 7.1 in 2022.

What is Reckless Abandon?

The proper way of upgrading a Laravel application is typically to keep it upgraded over time so that no single upgrade is any major source of effort. If you are constantly updating your application and its dependencies, there's no reason not to work on framework upgrades along the way. What about applications that don't have the luxury of being well enough funded to have a development staff working on it full-time? What about those that don't have part-time staff!? These applications sit on (hopefully) secure systems receiving patched versions of PHP. This particular application currently lives in Microsoft Azure on an LTS operating system with LTS PHP versions so while we're running ancient versions there is continued support and patches coming. What is far more common is that you have these kinds of applications running on unmanaged infrastructure, which ends up becoming a 0-day security liability. There is more reckless abandon in leaving an unpatched Linux server exposed to the internet than the blatant version hurdling we're about to put our application through.

How Did We Get Here

We built our application over a weekend during a charity hackathon, GiveCamp Memphis[1], for a small nonprofit that provides relief to those suffering long-term illnesses, which is typically a gift card. Built over a weekend hackathon focused on helping local nonprofit organizations, the application we're upgrading has one major pain point: A monster HTML form with many fields. The application scope was laser focused on providing an application to present users with a large form to fill out and submit the form. User experience wasn't even a consideration because the process we were replacing was a paper-based manual request system based on the United States Postal Service. Prospective recipients would send a letter, and the organization would send out donations to those that could be validated. Unfortunately, there always seems to be someone running a scam, so we needed the application to track incoming form submissions, which would be referrals. The application administrator could view the pending referrals and update/edit/sort and mark them with various statuses. We have a massive form with a CRUD (Create, Read, Update, Delete records) application built around it. And it was built in less than 72 hours. But it has tests!

The application's primary focus is the monster referral creation form containing 15 fields, including mostly text inputs and one file upload field. The form is somewhat clean since the application used `laravelcollective/html` package, which includes Form field creation helpers, such as opening our form is `{!! Form::open(['route' => 'referrals.create', 'files' => true, 'class' => 'form-horizontal']) !!}`, instead of a traditional HTML `<form ...`. We also use the package to create our file upload: `{!! Form::file('verification', ['class' => 'btn btn-default']) !!}`. This package will need to come along with our upgrade, or else we'll have to revert back to using HTML if we can't use the `laravelcollective/html` package.

Our application also uses Laravel Scout and a custom MySQL scout driver, which we'll need to ensure continues to work post-upgrade. We use Scout to allow for referral searching on the administrative side. We're using `league/csv` to provide export functionality, and we use `laravel/browser-kit-testing` to HTTP test our forms. All of the confidence in our tests relies on the BrowserKit Testing package, which allows us to make easy assertions based on output. If the application used a fancy front-end framework, we would

[1] https://givecampmemphis.org

have to use a traditional browser-based testing framework, which would be much slower and brittle. The most expensive part of our test suite is we're relying on MySQL to be present.

What's Between Here and There?

Before starting this process, what's in the Laravel Framework between our current version and the latest? Authentication and the underlying scaffolding have been completely revamped and improved—date serialization, CORS changes, model factories, pagination, PHP return types, and major dependency updates are just a sampling of the "High" and "Medium Impact Changes" in the Laravel Upgrade guides. We're going to skip 5.7 and 5.8, which bought small but valuable changes to environment variable parsing, notification channels, and removing the Blade `or` operator.

Laravel 6 brought us the last denoted "LTS" or Long Term Support version in September 2019. Laravel 6 also moved all the `str_` and `array_` helper methods to the `laravel/helpers` package. This is one change that will likely impact us, but we'll add the helpers' package to our new application to ensure an easy transition.

Laravel 7 added a Guzzle-wrapped HTTP client, markdown mail improvements, and MySQL 8 queue enhancements. We also saw authentication and related scaffolding moved to the `laravel/ui` package and required PHP 7.2.5 as a minimum. The upgrade guide offers that a version 6 to 7 upgrade should take about 15 minutes. Unless you're overriding a lot of the Symfony-provided methods that were updated to version 5 during this time, you likely wouldn't have trouble with this upgrade. Mail configuration also changed, which might cause some hiccups during the upgrade process. Our application only sends one email from a blade view, so any changes should be negligible.

Laravel 8 was released on September 8, 2020, and included Jetstream, model factory classes, migration squashing, and many other improvements. Jetstream[2] is a starter kit for Laravel that uses TailwindCSS[3] and offers Livewire or Inertia JavaScript scaffolding. If you enjoy learning by example, Jetstream is a fantastic resource on how to build out elegant front ends with Laravel tooling and TailwindCSS. Upgrading from 7 to 8 is listed as another 15-minute migration, depending on how you've built your application. PHP 7.3.0 is the minimum requirement so we're sailing right on by those PHP versions on our way to 8.1.

Laravel 9 continues the yearly Laravel release cadence starting with version 8. The major change was moving to Symfony 6.0 components, Symfony Mailer, Flysystem 3, and a Laravel Scout database driver. The Scout database driver will allow us not to worry about the MySQL driver package we previously relied on, which means we've eliminated a dependency. Laravel 9 also brings the minimum PHP version to 8.0. The upgrade from 8 to 9 is listed to be 30 minutes, with most of the time spent updating composer dependencies to PHP 8+ compatible versions. Laravel 9 also is where the framework starts to use PHP return types, which won't affect our application as we're not overriding any of Laravel's core classes. If you are overriding and extending core classes, note that you may start running into speed bumps or even potholes with Laravel 9.

Inspecting the Application

Our application is the typical Laravel smattering of routes pointing to controller methods that return Blade views. The front end is Bootstrap 4 and extended many of the default Laravel authentication bootstrap styles of the Laravel 5.6 era. The application models the referrals, donors, and gifts, a referral with a special honorarium or message attached. The public routes are limited to authentication, creating referrals, and administrative pages. (See Listing 1)

Listing 1.

```
use Auth\LoginController;
Route::get('login', 'LoginController@showLoginForm')
      ->name('login');
Route::post('login', 'LoginController@login');
Route::get('logout', 'LoginController@logout')
      ->name('logout');
Route::get('/', [
         'as' => 'home',
         'uses' => 'HomeController@index'
]);
Route::post('/referrals', [
     'as' => 'referrals.create',
     'uses' => 'ReferralController@create'
]);
Route::group(['middleware' => ['auth']], function () {
    Route::get('/referrals', [
        'as' => 'referrals.index',
        'uses' => 'ReferralController@index'
    ]);
    Route::get('/referrals/fulfilled', [
        'as' => 'referrals.index.fulfilled',
        'uses' => 'AdminController@getFulfilled'
    ]);
    Route::get('/referrals/unfulfilled', [
        'as' => 'referrals.index.unfulfilled',
        'uses' => 'AdminController@getUnfulfilled'
    ]);
    Route::get('/referrals/archived', [
        'as' => 'referrals.index.archived',
        'uses' => 'AdminController@getArchived'
        ]);
    Route::get('/admin', [
        'as' => 'admin.index',
        'uses' => 'AdminController@index'
    ]);
...
```

[2] Jetstream: *https://jetstream.laravel.com*

[3] TailwindCSS: *https://tailwindcss.com*

We're using the auth middleware to protect the sensitive parts of the application, and we do not allow users to register, which means the only users are administrators. In traditional applications, it makes sense to at least have users and administrators separately; however, this organization will only ever have one user, and there isn't much growth expected. Our goal isn't to be the biggest or brightest; we want to provide functionality that solves the user's problem.

To run our test suite, we'll run ./vendor/bin/phpunit and see if everything is working as expected.

```
vagrant@homestead:~/app$ ./vendor/bin/phpunit
PHPUnit 7.0.1 by Sebastian Bergmann and contributors.

.......                                    7 / 7 (100%)

Time: 1.75 seconds, Memory: 22.00MB

OK (7 tests, 28 assertions)
```

An example of the type of test contained in the application that gives us so much confidence would be the referral form test. (See Listing 2)

Listing 2.

```
public function testCreateReferral()
{
  $user = factory(App\User::class)->create();

  $this->actingAs($user)
    ->withSession(['foo' => 'bar'])
    ->visit('/referrals')
    ->see('Request Gift Card')
    ->type('Recipient Name', 'recipient_name')
    ->type('recipient@email.com', 'recipient_email')
    ->type('Friend', 'patient_relationship')
     ... removed to fit this space, see downloads
    ->type('Treatment Location', 'facility_name')
    ->type('Gift Card Type', 'giftcard_type')
    ->attach(public_path().'/logo.jpeg','verification')
    ->press('Submit Request')
    ->seeInDatabase('referrals', [
      'recipient_name' => 'Recipient Name',
      'recipient_email' => 'recipient@email.com',
      'submitter_name' => 'Submitter Name',
      'submitter_email' => 'submitter@email.com',
      'patient_relationship' => 'Friend',
      'address' => '1234 somewhere street',
      'address2' => '44th floor',
      'city' => 'Some City',
      'state' => 'TN',
      'zip_code' => '90210',
      'cancer_type' => 'Illness Type',
      'doctors_name' => 'Doctors Name',
      'facility_name' => 'Treatment Location',
      'giftcard_type' => 'Gift Card Type',
    ])
    ->dontSee('Whoops');
}
```

Another test uses a factory to create an initial referral and attempts to update it: (See Listing 3)

Listing 3.

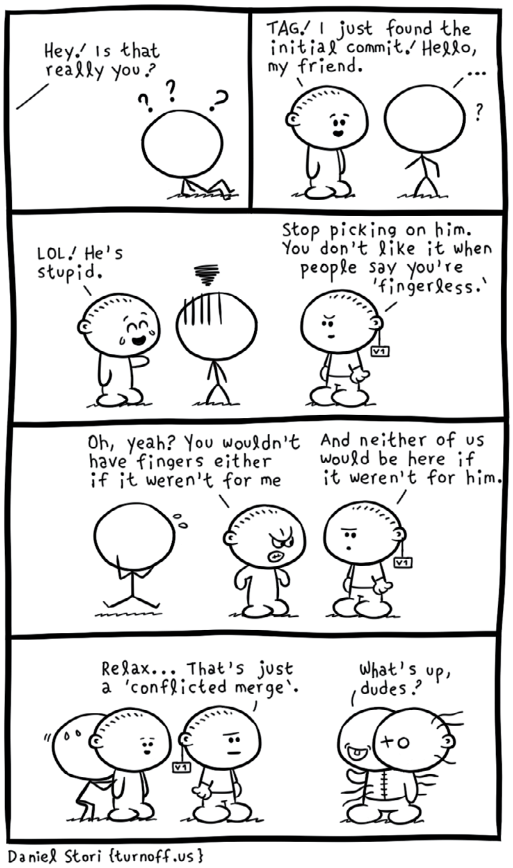
```
public function testUpdateReferral()
{
  $user = factory(App\User::class)->create();
  factory(App\Referral::class, 2)->create();
  $referral = Referral::all()->random();

  $this->actingAs($user)
    ->withSession(['foo' => 'bar'])
    ->visit('/referrals/' . $referral->id . '/edit')
    ->see('Edit Referral')
    ->type('Updated Recipient Name', 'recipient_name')
    ->type('up_recipient@email.com', 'recipient_email')
    ->type('Updated Submitter Name', 'submitter_name')
    ->type('up_submitter@email.com', 'submitter_email')
    ->type('Updated Friend', 'patient_relationship')
    ->type('Updated 1234 somewhere street', 'address')
    ->type('Updated 44th floor', 'address2')
    ->type('Updated Some City', 'city')
    ->type('Updated TN', 'state')
    ->type('Updated 90210', 'zip_code')
    ->type('Updated Illness Type', 'illness_type')
    ->type('Updated Doctors Name', 'doctors_name')
    ->type('Updated Treatment Loc', 'facility_name')
    ->type('Updated Gift Card Type', 'giftcard_type')
    ->press('Update Referral')
    ->seeInDatabase('referrals', [
      'id' => $referral->id,
      'recipient_name' => 'Updated Recipient Name',
      'recipient_email' => 'up_recipient@email.com',
      'submitter_name' => 'Updated Submitter Name',
      'submitter_email' => 'up_submitter@email.com',
      'patient_relationship' => 'Updated Friend',
      'address' => 'Updated 1234 somewhere street',
      'address2' => 'Updated 44th floor',
      'city' => 'Updated Some City',
      'state' => 'Updated TN',
      'zip_code' => 'Updated 90210',
      'cancer_type' => 'Updated Illness Type',
      'doctors_name' => 'Updated Doctors Name',
      'facility_name' => 'Updated Treatment Loc',
      'giftcard_type' => 'Updated Gift Card Type',
    ])
    ->dontSee('Whoops');
}
```

One characteristic of our tests you might notice, we have a small number of tests with a high number of assertions. Each assertion is our check that our fields are passing data from the front end to the backend properly. These tests, while brief, exercise large portions of the application's complexity. Testing failure scenarios and your successful workflows are also essential to ensure that when things go wrong, they fail in ways we expect them to. Another way to bolster your test suite is to add any new test cases *now* but resist the urge to refactor code. We'll have to tweak the code as we go because we are dodging Laravel changes and major PHP version changes.

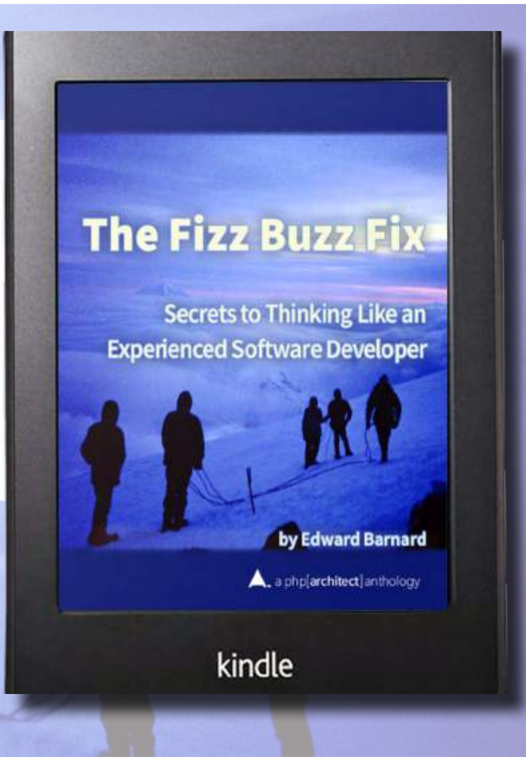
While a strong test suite gives us confidence in this style of wild west coding, a strong discipline of doing things the "framework" way, helps keep our code simple. Our referral application isn't going to reinvent the wheel or discover a new idea. We want to keep it simple (KISS!) and stick to conventions so that we can easily pick up where we left off when we come back to this (potentially in another few versions).

Next month we'll spin up a fresh application with our dependencies and start copying our files over. We'll watch as our test suite screams that everything is on fire and chaos is reigning in our application; it's going to be great. I hope this first part has inspired you to go write more tests!

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


Hey! Is that really you?
? ? ?
TAG! I just found the initial commit! Hello, my friend.
...
?
LOL! He's stupid.
Stop picking on him. You don't like it when people say you're 'fingerless.'
V1
Oh, yeah? You wouldn't have fingers either if it weren't for me
And neither of us would be here if it weren't for him.
V1
Relax... That's just a 'conflicted merge'.
What's up, dudes?
+o
Daniel Stori {turnoff.us}

turnoff.us | Daniel Stori
Shared with permission from the artist


The Fizz Buzz Fix
Secrets to Thinking Like an Experienced Software Developer
by Edward Barnard
a php[architect] anthology
kindle

Any Two Birthdays

Oscar Merida

Working with dates and times is fraught with peril, so it helps to practice manipulating such data. We'll have to be wary of how to calculate differences between dates, potentially deal with leap days, and other irregularities.

A Brief History of Calendars

While our planet orbits the sun without much regard for measuring its progress, timekeeping to measure the days of the year and the cycles of the sun goes back all the way to the paleolithic era. The earliest calendars were based on lunar and solar cycles, like Hindu calendars with 12 lunar months and 365 solar days. Because the 12 lunar months didn't quite align with the solar year, many had "leap months" to make up the differences.

The Julian calendar, introduced by the Romans in 45 BCE, did away with the lunar months, gave us the month names we still use, and also introduced a leap-day every four years. Calendar changes didn't stop there, however. Calendar reform in Persia in the 11th century measured the length in days of a year to an astonishing accuracy (365.24219858156 days) and the Gregorian calendar refined the Julian calendar in 1582.

Keep in mind that while most of us only have to consider one calendar, there are other calendaring systems[1] still in use around the world.

Recap

> *Write a program that takes two people's birthdays and outputs the following information for the two dates:*
>
> *Were they born on the same day of the week?*
>
> - *Are they born in the same meteorological season?*
> - *Were they both born on an even or odd day?*
> - *What is their difference in ages in years, months, and days?*
> - *In a given calendar year, how many days are between their birthdays?*

Parsing Input

A stumbling block for working with dates is parsing datetime input and representing it in a way that computers can use. An all-to-common approach is to parse the dates into UNIX timestamps and then add/subtract seconds to calculate new dates or differences between them. For small ranges, this can work—until you end up calculating on a day when Daylight Savings time kicks in or on a leap day.

Luckily for us, PHP has the awesome and capable Datetime library[2] by Derick Rethans. If you need to parse dates, calculate differences, compare dates, or manipulate them in any way, you should be using it. It also handles timezones, conversions, and accounts for changes like DST and leap days. Let's assume we can work with any birthday date string parseable by the library.

```
$jose = new \DateTimeImmutable('25 March 1946');
$sandra = new \DateTimeImmutable('6/16/2000');
```

If we peek at the internals of each object we'll see it converted them and used the default time zone set via php.ini. Since we didn't specify a time, it defaults to midnight. See Listing 1.

Listing 1.

```
object(DateTimeImmutable)#1 (3) {
  ["date"]=>
  string(26) "1946-03-25 00:00:00.000000"
  ["timezone_type"]=>
  int(3)
  ["timezone"]=>
  string(3) "UTC"
}
object(DateTimeImmutable)#2 (3) {
  ["date"]=>
  string(26) "2000-06-16 00:00:00.000000"
  ["timezone_type"]=>
  int(3)
  ["timezone"]=>
  string(3) "UTC"
}
```

The DateTime library will do its best to parse dates it gives you, according to your configured locale. For example, the following is interpreted as "May 8th, 2022" for me, but if you use a more sane date/month order, it might be parsed as "August 6th, 2022". If it can't parse an input string, the DateTime library will throw an `\Exception`.

```
$eric = new \DateTimeImmutable("5/8");
```

[1] *other calendaring systems: https://phpa.me/worldatlas*

[2] *Datetime library: https://php.net/Datetime*

Same Day of Week

Now that we have Jose and Sandra's birthdays as DateTime objects, we can use the library's methods to answer our questions. Were they born on the same day of the week? See Listing 2.

Listing 2.

```
echo "Jose was born on " . $jose->format('l') . PHP_EOL;
echo "Sandra was born on " . $sandra->format('l') . PHP_EOL;

if ($jose->format('D') === $sandra->format('D')) {
    echo "They were born on the same day!" . PHP_EOL;
} else {
    echo "They were not born on the same day.".PHP_EOL;
}
```

Gives the following output:

```
Jose was born on Monday
Sandra was born on Friday
They were not born on the same day.
```

Meteorological Season

Another calendar quirk to deal with is that weather forecasters and astronomers debate when the seasons start. Most calendars mark the start of each season based on the summer and winter solstice and spring and fall equinoxes. These are the days with the longest or shortest amount of daylight, or equal amounts of daytime and nighttime. These astronomical events can shift by a day or two from year to year. Luckily for this question, the meteorological seasons are contained to months.

- **Spring:** March, April, May
- **Summer:** June, July, August
- **Fall:** September, October, November
- **Winter:** December, January, February

We can write a straightforward function to return the season given an input date. See Listing 3.

Then we can output what season Jose and Sandra were born.

```
echo 'Jose was born in the ' . getSeason($jose);
echo 'Sandra was born in the ' . getSeason($sandra);
// Jose was born in the spring
// Sandra was born in the summer
```

Odd or Even

Likewise, we can use a simple to test if both were born on an odd or even day. Since the question doesn't ask us to determine if the day was odd or even, we can use the following hack. This solution does not require that we test each number for odd/even-ness. See Listing 4.

Listing 3.

```
function getSeason(\\DateTimeImmutable $date): string
{
    switch ($date->format('M')) {
        case 'Mar': case 'Apr': case 'May':
            return 'spring';
        case 'Jun': case 'Jul': case 'Aug':
            return 'summer';
        case 'Sep': case 'Oct': case 'Nov':
            return 'fall';
        case 'Dec': case 'Jan': case 'Feb':
            return 'winter';
        default:
            return '';
    }
}
```

Listing 4.

```
// get numeric day and add them together
$sum = (int) $jose->format('j') +
       (int) $sandra->format('j');

// if the sum is even, both were born on
// either odd or even days
$isEven = (0 === $sum % 2);
if ($isEven) {
    echo 'Both were born on an odd or even day';
} else {
    echo 'One was born on an odd day, ' .
         'the other on an even day.';
}
```

Difference in Ages

Since we have lovely DateTimeImmutable objects, we can use the built-in `diff()` method to find the difference. Doing so is preferable to converting them to seconds and then turning that back into a human-readable string. If you had to, you could also compare dates in different timezones, per the docs:

> *The method is aware of DST changeovers, and hence can return an interval of 24 hours and 30 minutes, as per one of the examples. If you want to calculate with absolute time, you need to convert both the this/baseObject, and $targetObject to UTC first.*

```
$diff = $jose->diff($sandra);
var_dump($diff);
```

Shows a object that begins with: See Listing 5.

Each property refers to the amount of years, months, days, and even hours, minutes, and seconds difference between the dates. Since we only have dates, we can ignore the time components for our output.

Listing 5.

```
["y"]=>
int(54)
["m"]=>
int(2)
["d"]=>
int(22)
["h"]=>
int(0)
["i"]=>
int(0)
["s"]=>
int(0)
```

```
echo "The difference is " . $diff->y . ' years, '
    . $diff->m . ' months, and ' . $diff->d . ' days.';
// The difference is 54 years, 2 months, and 22 days.
```

Days Between

To figure out the calendar days between birthdays, we first have to get a DateTime object of the dates in the same year. We can use the constructor to do that work for us. You might be tempted to use `diff()` again, but it "rolls up" the difference into chunks of year, months, and days. Instead, we can use the z format specifier, which gives the ordinal day of the year, between `0` and 365, for a specific date. Once we have the ordinal day for each date, we can readily calculate the absolute number of days between them.

> *Note: I think in a leap year, z should return a value up to 366, but the php.net documents list the max as 365. See Listing 6.*

Stats 101 Grade Book

> *Given the 40 grades for a course below, find the mean, median, and mode for all grades. Then plot a histogram of letter grades, ignoring pluses and minuses, given the indicated scale.*

Grades:

```
91, 86, 70, 81, 92, 80, 73, 85, 70, 87, 74, 82, 77, 83,
90, 90, 87, 83, 93, 72, 84, 87, 83, 73, 86, 81, 86, 77,
75, 89, 77, 80, 79, 95, 69, 78, 89, 84, 70, 72, 89,
```

Letter Grade Assignment

Letter	Percentage	Letter	Percentage
A+	97-100%	C	73-76%
A	93-96%	C-	70-72%
A-	90-92%	D+	67-69%
B+	87-89%	D	63-66%
B	83-86%	D-	60-62%
B-	80-82%	F	0-59%
C+	77-79%		

Listing 6.

```
$jose22 = new \DateTimeImmutable(
                $jose->format('2022-m-d')
          );
$sandra22 = new \DateTimeImmutable(
                $sandra->format('2022-m-d')
            );

$daysDiff22 = abs(

    (int)$jose22->format('z') - (int)$sandra22->format('z')
);

echo "There are " . $daysDiff22 . ' days between ' .
     'their birthdays in a calendar year.';
There are 83 days between their birthdays in a calendar year.
```

Some Guidelines And Tips

The puzzles can be solved with pure PHP. No frameworks or libraries are required.

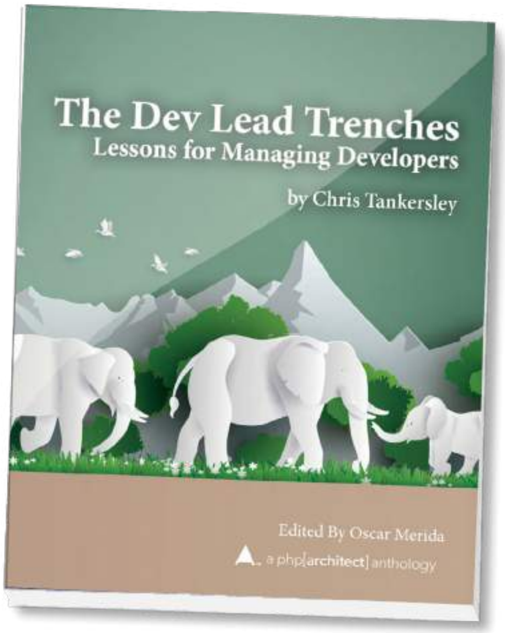
- *Each solution is encapsulated in a function or a class, and I'll give sample output to test your solution against.*
- *You're encouraged to make your first attempt at solving the problem without using the web for clues.*
- *Refactoring is encouraged.*
- *I'm not looking for speed, cleverness, or elegance in the solutions. I'm looking for solutions that work.*
- *Go ahead and try many solutions if you like.*
- *PHP's interactive shell (`php -a` at the command line) or 3rd party tools like PsySH*[3] *can be helpful when working on your solution.*
- *To keep solutions brief, we'll omit the handling of out-of-range inputs or exception checking.*

Related Reading

- *PHP Puzzles: The Birthday Paradox* by Oscar Merida, July 2022. https://phpa.me/puzzles-july-2022
- *PHP Puzzles: Fractional Math by Oscar Merida, September 2022.* https://phpa.me/puzzles-sep-2022

Oscar Merida has been working with PHP since version 4 was released and is constantly learning new things about it and still remembers installing an early version on Apache. When not coding or writing, he enjoys RPGs, soccer, and drawing. @omerida

[3] PsySH: *https://psysh.org*



Try, or Try Not; There is no Do

Edward Barnard

The past two months focused on transactional-consistency boundaries. This month we're extending the concept to RESTful API requests and responses. We'll ensure that responses are internally consistent, whether success response, or error response. We'll carefully draw and enforce the consistency boundary using the ancient wisdom of Yoda.

Transactional Boundary

> *Do, or do not; there is no try. –Yoda, discussing PHP 4*

The previous two "DDD Alley" articles, "Transactional Boundary" and "Exception Report", introduced the idea of a transactional boundary. The transactional boundary is "all or nothing", meaning all state changes must succeed or all fail. We don't want to leave database tables in an inconsistent state.

Consider, for example, an invoice containing line items. The invoice body resides as a row of the `invoices` table. The invoice line items reside as rows of the `invoice_line_items` table. The invoice total is part of the invoice body and is a sum of the line-item amounts.

Let's follow Yoda's advice to "just do it". The user story, this iteration, says we need to update each line item with a changed price. You will recall that our procedure is to do all database updates within a database transaction. Place that transaction within a try/catch. But we're not following that procedure! Instead, we're blindly following Yoda's advice to "just do it" (no try, no catch, no database transaction).

Something unfortunate happened. One of the price changes was missing—three line items got updated, while the fourth line item wasn't updated. We completely forgot to update the invoice total since our user story did not mention that requirement.

The invoice is now in an inconsistent state. That's a problem! We correctly detected the missing price change and correctly did NOT update that line item. We definitely implemented Yoda's advice ("do", and for the missing price change, "do not").

Yoda's 1980 advice from *Star Wars: Episode V—The Empire Strikes Back* is, I'm sorry to say, sadly out of date. PHP 5 brought us a new hope, so to speak, by introducing exceptions and `try/catch`. The correct advice–and this remains true with PHP Episode VIII–is "try, or try not; there is no do".

In fact, the database transaction is not important; it's merely an implementation detail. What's important is the "all or nothing" concept of maintaining internal consistency.

Request-response Integrity

Let's return to the modern era. We have invoices. They are internally consistent. We spent considerable "whiteboard" time walking through scenarios with our stakeholders. We created a concise set of invariants, that is, business rules which must remain true throughout the life cycle of the "thing".

We carefully drew transaction boundaries. We placed anything that must remain continuously and immediately consistent within that boundary. We provided for eventual consistency *outside* the transaction. Our stakeholders provided an exact timeframe for "eventual" consistency. The ensuing lack of errors felt great, but we kept that observation to ourselves.

We have a new iteration. And a fresh user story. We are building out an API-First[1] back end. Our new task is to provide a specified invoice with body and line items (a RESTful http GET request and response). All we're doing is retrieving the invoice for display.

Of course, it's never quite that simple! That's why we're here. In order to make the request, our client (the user or application consuming our API) must be logged in (authenticated) and allowed to view this specific invoice (authorized). We'll use the http response-status codes to indicate the situation:

- `401` if not logged in
- `403` if logged in but not allowed to see this invoice
- `404` if the requested invoice does not exist
- `400` if the request is invalid, such as a malformed invoice number
- `200` with the response body containing the requested invoice

Suppose that something went wrong as we were fetching the invoice. Suppose we returned the invoice body but forgot to include its line items. The returned invoice is not internally consistent. I've seen this sort of thing happen with third-party APIs where not all fields are populated as expected. Consistency boundaries are important!

Generally speaking, here's what we want to do. We want to create an "all or nothing" response. We either return the entire invoice, internally consistent, with a `200` status, or we return an error response with some other http status indicating the error. The error status does not contain any portion of the requested invoice.

[1] https://www.postman.com/api-first/

Do you see that we've just created a consistency boundary? Our design does not require a database transaction. We're not changing the state of the system—we're simply reporting the existing state of an existing invoice. It's a "read only" operation. From the *database* perspective, nothing changes; therefore, no consistency rules apply.

However, from the *client* perspective, we do have consistency rules. We expect the client to either: receive the requested invoice in full and internally consistent (status `200`) or an error response telling the client what went wrong (such as status `404`).

Our design/implementation strategy will be:

- Surround the request/response processing with a `try/catch` block.
- The "happy path" is when everything works, and we have retrieved the invoice. Respond with `200` status and the requested invoice.
- For all other situations, throw an `Exception`, which causes `catch` block execution.
- Inside the `catch` block, respond with the appropriate error response (see below).

The happy path is easy. We simply respond with status `200` and the suitably-formatted invoice. Let's consider how to deal with the error responses.

Ctgo Response Object

The API-First application I'm currently building is called Clay Target Go![2], or "ctgo" for short. We're about to design a custom response object that applies to the application as a whole, thus the name "ctgo response object".

The problem is this: We reach the `catch` block because something went wrong. We need to figure out *which* status code to return with the error response. One approach might be to `throw` different exceptions for different situations, with each type of exception corresponding to a specific http response code. However, I suspect that would lead to a lot of code bloat. Every time we process an API request-response, we wrap the processing with `try` and several different `catch()` blocks. Furthermore, each `catch` block is merely guessing based on having received an exception.

I decided to construct the failure response at the source. That's the `CtgoResponse` purpose:

> *The CtgoResponse object is primarily aimed at passing error information back to the controller or Application Service from inside a transaction. This process goes hand-in-hand with CtgoException. The transaction is inside a try/catch, and with a separate catch block for CtgoException, the error information can be passed back via throwing the exception.*
>
> *The controller (and the api description yaml) should know the possible response codes for a given request and therefore what to do with the CtgoResponse object (serialize, redirect, etc.).*

In other words, `CtgoResponse` is a container for error information. See Listing 1[3].

Listing 1.

```php
final class CtgoResponse
{
    private int $statusCode;
    private string $statusText;
    private string $errorSummary;
    private string $errorDescription;
    private array $responseBody;

    public function __construct(
        int $statusCode = 200,
        string $statusText = 'OK',
        string $errorSummary = '',
        string $errorDescription = '',
        array $responseBody = []
    ) {
        $this->statusCode = $statusCode;
        $this->statusText = $statusText;
        $this->errorSummary = $errorSummary;
        $this->errorDescription = $errorDescription;
        $this->responseBody = $responseBody;
    }

    public function getStatusCode(): int {
        return $this->statusCode;
    }

    public function getErrorResponseBody(): array {
        return [
            'status_code' => $this->statusCode,
            'status_text' => $this->statusText,
            'error_summary' => $this->errorSummary,
            'error_description' =>
                $this->errorDescription,
        ];
    }

    public function getResponseBody(): array
    {
        return $this->responseBody;
    }

    public function getStatusText(): string
    {
        return $this->statusText;
    }
}
```

[2] Clay Target Go!: http://mnclaytarget.com/team-management/

[3] Listing 1: https://phpa.me/responseobject

Ctgo Exception

The response object container requires a container of its own:

> *The CtgoException allows us to pass error information back to the caller from inside a database transaction.*

See Listing 2. The act of “throwing” an `Exception` normally creates a new `Exception` class (or subclass such as `CtgoException`) object, with information passed into the constructor. Both `CtgoResponse` and `CtgoException` are simply containers designed to capture information via their constructors

Listing 2.

```php
<?php
class CtgoException extends RuntimeException
{
    private CtgoResponse $ctgoResponse;

    public function __construct(
        CtgoResponse $ctgoResponse,
        string $message = '',
        int $code = 0,
        ?Throwable $previous = null
    ) {
        parent::__construct($message, $code, $previous);
        $this->ctgoResponse = $ctgoResponse;
    }

    public function getCtgoResponse(): CtgoResponse
    {
        return $this->ctgoResponse;
    }
}
```

Controller Method

How do we implement the “all or nothing” boundary? See Listing 3.

Lines 12-14 contain our request processing. In this case, we’re providing details on a team’s head coach rather than for an invoice.

At line 12, we verify that this client is authorized to make this request. If not, the authorization code will create a `CtgoResponse` object containing the appropriate error response.

Listing 4 shows a snippet of the request authorization. If the client’s login session is expired, assemble an error response with status `403`. `CtgoResponse` contains the error response, and `CtgoException` contains `CtgoResponse`. Throwing `CtgoException` sends us to the `catch()` block, line 15 of Listing 3. We’ll examine the `catch` block below.

Line 13 of Listing 3 loads information concerning the team in question. The authorization step already needed to

Listing 3.

```php
 1. <?php
 2. class TeamsController extends AppController
 3. {
 4.     public function viewHeadCoach(): ?Response
 5.     {
 6.         $this->request->allowMethod('get');
 7.         $ctgoAuthorize =
 8.             CtgoAuthorizeFactory::forTeamProfile(
 9.                 $this->request
10.             );
11.         try {
12.             $ctgoAuthorize->authorize();
13.             $team = $ctgoAuthorize->getTeam();
14.             $userRole = $this->loadUserRole($team);
15.         } catch (CtgoException $e) {
16.             $this->processCtgoException($e);
17.             return null;
18.         }
19. 
20.         $results = (new MapTeamStaff())->map($userRole);
21. 
22.         $this->set(compact('results'));
23.         $this->viewBuilder()
24.             ->setOption('serialize', 'results');
25. 
26.         return null;
27.     }
28. }
```

Listing 4.

```php
<?php
final class ForTeamProfile extends CtgoAuthorize
    implements CLookupRoleTypes
{
    public function authorize(): void
    {
        $this->loginToken = $this->repository
            ->loadLoginToken();

        $this->team = $this->repository
            ->loadTeamWithLeague();

        $examiner = new ExamineToken($this->loginToken);
        if ($examiner->isExpired()) {
            $response = new CtgoResponse(403);
            throw new CtgoException(
                $response, 'Login expired', 403
            );
        }
        /* ...additional code here... */
    }
}
```

examine that information, so that step memoized[4] the result and provides it here.

Line 14 loads the desired head coach information. If anything goes wrong (such as resource not found), method `$this->loadUserRole($team)` throws a `CtgoException` containing the appropriate `CtgoResponse`.

At this point (i.e., upon safely completing line 14), the requested information `$userRole` is in "database" format rather than "API response" format. At line 20, `MapTeamStaff::map()` converts the response to the necessary format. Lines 22-27 tell the CakePHP 4.x framework to generate the `application/json` response.

Catch Block

Lines 16-17 of Listing 3 are the `catch` block. When anything goes wrong, processing should reach these two lines of code. The `catch` processing, `processCtgoException()`, is in the parent class. See Listing 5.

Method `processCtgoException()` merely copies information from the CtgoResponse container to the framework's response object. We already built the response, complete with status code, at the point where we encountered the error.

Listing 5.

```php
<?php
class AppController extends Controller
        implements CApiCodePrefix
{
    protected function processCtgoException(
        CtgoException $e
    ): void {
        $response = $e->getCtgoResponse();
        $statusCode = $response->getStatusCode();
        $statusText = $response->getStatusText();
        $success = $response->getErrorResponseBody();
        $this->set(compact('success'));
        $this->viewBuilder()
            ->setOption('serialize', 'success');
        $this->response = $this->response
            ->withStatus($statusCode, $statusText);
    }
}
```

Note that this method processes *error* responses, not "happy path" responses. Various "happy paths" could have a large variety of possible responses, whereas we have a limited set of standardized error responses.

Designing APIs with Swagger and OpenAPI[5] by Joshua S. Ponelat and Lukas L. Rosenstock (2022), page 310, provides some guidelines for making error responses developer friendly:

- An error response should be clearly distinguishable from a successful response.
- Both success and error responses should have the same data serialization format.
- The data structure (the JSON schema) should be similar for all error responses.
- Having a common and consistent structure helps the API consumer because they can reuse more of their error-handling code.

Designing APIs further recommends that the error information include human-readable explanations aimed at helping the client's developers understand what went wrong and why—which should assist with more rapidly resolving the problem.

What About Updates?

We've been examining http GET requests. They do require internal consistency, but they are still "read only" requests that do not change the state of the system. Therefore, we did not need to consider database transactions as part of our implementation.

However, our application's RESTful API does include "create" and "update" requests (http POST and PATCH). Those requests do require database transactions to enforce atomic consistency. This observation, that reads do not use database transactions, while creates and updates do, leads me to consider the CQRS pattern.

Martin Fowler describes the CQRS[6] pattern with an important warning:

> *CQRS stands for Command Query Responsibility Segregation. At its heart is the notion that you can use a different model to update information than the model you use to read information. For some situations, this separation can be valuable, but beware that for most systems CQRS adds risky complexity…*
>
> *The change that CQRS introduces is to split that conceptual model into separate models for update and display… the rationale is that for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that does neither well.*

We know that we want any API response to be internally consistent, regardless of whether it's a "query" (read) or "command" (create/update/delete) request. In fact:

- We should be able to easily distinguish between success and failure responses.

[4] https://en.wikipedia.org/wiki/Memoization

[5] https://phpa.me/swaggerandopenapi

[6] https://martinfowler.com/bliki/CQRS.html

- Each response, whether success or failure, must be internally consistent.

For “read” requests, we wrap request processing in a top-level `try/catch` block. We design all request processing to throw a `CtgoException` when the response should be an error response. That top-level `catch()` generates the suitable response via `processCtgoException()`.

For “command” requests, we need to add one more step. Within that top-level `try/catch` block, use a top-level database transaction. See Listing 6.

I decided to maintain two separate `Controller` classes, one for Query requests and one for Command requests. That’s because I implement the top-level consistency-boundary differently. If I mix them all into one file, I might easily copy/paste the wrong type of boundary. To me, it makes sense to separate Command and Query API handlers, even though they might both make use of the same internal domain models.

Lines 6-16 are essentially identical to their Query implementation. That’s why I kept this part, authentication, as a separate `try/catch` block. Lines 18-46 are the actual request processing.

Lines 19-37 contain the database transaction, the “all or nothing” database consistency boundary. The framework ORM[7] includes a “save or fail” method, which throws an exception upon failure. If something goes wrong, we *want* to fail.

Note line 38. Anything which does not belong inside the database transaction, but is part of our request/response consistency requirement, could go here. At this point, we could queue email for sending, notify interested parties of domain events, and so on.

At line 39, we process the `CtgoException`. This `catch` is for *known* errors. Anything else, such as a database error, will fall through to line 42 and generate a response with status `400` (badly-formed API request).

Finally, if all went well, we return status `204` (no content). No response body is needed when updating a resource, and the resource was successfully updated according to the request.

Summary

Yoda, sure enough, expressed things backwards.

> *Try, or try not; there is no do.*

We learned how to use `try/catch` to carefully separate a success response from an error response. “Try” encompasses the happy path, the success response. “Try not”, the `catch()` processing, provides consistent handling of all non-happy paths. Don’t just “do it”; that is, process the request without guaranteeing internal consistency.

Listing 6.

```
 1. <?php
 2.
 3. class UpdateTeamsController extends AppController
 4. {
 5.   public function updateSchool(): ?Response
 6.   {
 7.     $this->request->allowMethod('patch');
 8.     $ctgoAuthorize =
 9.       CtgoAuthorizeFactory::forTeamProfileUpdate(
10.         $this->request
11.       );
12.     try {
13.       $ctgoAuthorize->authorize();
14.     } catch (CtgoException $e) {
15.       $this->processCtgoException($e);
16.       return null;
17.     }
18.
19.     try {
20.       $this->Teams->getConnection()
21.         ->transactional(function ()
22.         use ($ctgoAuthorize) {
23.           // for modified_by
24.           $loginToken = $ctgoAuthorize->getLoginToken();
25.           // Refresh team entity inside transaction
26.           $team = $this->Teams->get(
27.             $ctgoAuthorize->getTeam()->id
28.           );
29.           $team->modified_by = $loginToken->user_id;
30.
31.           $value = $this->request
32.                         ->getData('athletic_director');
33.           if (
34.             null !== $value &&
35.             (strlen($value) <= 250)
36.           ) {
37.             $team->athletic_director_name = $value;
38.           }
39.
40.           $this->Teams->saveOrFail($team);
41.         });
42.       // Other "eventually consistent" items here
43.     } catch (CtgoException $e) {
44.       $this->processCtgoException($e);
45.       return null;
46.     } catch (Exception) {
47.       return $this->response->withStatus(400);
48.     }
49.
50.     return $this->response->withStatus(204);
51.   }
52. }
```

Over the past two “DDD Alley” articles, “Transactional Boundary” and “Exception Report”, we learned that it’s absolutely important, while also being quite difficult, to design the correct consistency boundary in the correct place.

[7] https://phpa.me/saving-data

We need a similar boundary to protect the API request (and its response) as a whole. We saw how to use a top-level `try/catch` to implement that boundary. Drawing the boundary was easy–or was it?

On the contrary, that boundary got drawn during API design. Yoda showed us how to protect a *single* request's internal consistency. What if a series of API requests must be atomic, remaining consistent at all times? In that case, our API design must change. Those elements requiring atomic consistency must be contained in the same API request. Given that constraint, our boundary implementation holds.

Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others. *@ewbarnard*

Related Reading

- *DDD Alley: Random and Rare Failures* by Edward Barnard, June 2022. https://phpa.me/2022-06-ddd
- *DDD Alley: Structure by Use Case* by Edward Barnard, July 2022. https://phpa.me/2022-08-ddd
- *DDD Alley: Exploring Boundaries* by Edward Barnard, August 2022. https://phpa.me/ddd-aug-2022
- *DDD Alley: Domain Event Walkthrough* by Edward Barnard, September 2022. https://phpa.me/ddd-sept-2022
- *DDD Alley: Application Event Walkthrough* by Edward Barnard, October 2022. https://phpa.me/ddd-oct-2022
- *DDD Alley: Transactional-Boundary* by Edward Barnard, November 2022. https://phpa.me/ddd-nov-2022
- *DDD Alley: Exception Report* by Edward Barnard, December 2022. https://phpa.me/ddd-dec-2022

PSR 14: Event Dispatcher

Frank Wallen

As the title states, what PSR-14 is truly about is an Event Dispatcher. This is what sends out the Event into the system. We're also going to talk about Events and Listeners, as the dispatcher would have nothing to do without them.

In modern libraries and frameworks, Events are an invaluable workhorse, a simple object that carries a payload of data to be retrieved by systems and objects listening and waiting to respond to that Event. Systems or objects that react to specific events are usually called Listeners or Subscribers, and they perform tasks in response to the Event. For example, imagine an application where users buy tickets for various movie theaters. The user selects their movie, then the seats in the theater, and then submits payment for processing. The user expects to receive a QR code on their phone and in their email. At the end of the purchase, two events are sent out: one that will trigger the generation of an email and another that will trigger a notification sent to the user's phone. Other events may be triggered, such as notifying the theater that the user purchased the tickets too. Eventually, other supporting services could be added to the application such as promotional gifts for those seeing a particular movie that may be mailed to their home, or be waiting for them at the theater. Instead of building in additional code, a new event is created to be emitted. Very little code needs to be added in this case, just a new event, and a listener (of course, there must be a system that handles the moviegoer's promotional gift).

The full description of the PSR is located here[1], and the GitHub repository for the interfaces is located here[2]. As the title states, what PSR-14 is truly about is an Event Dispatcher. This is what sends out the Event into the system. We're also going to talk about Events and Listeners, as the dispatcher would have nothing to do without them. First, here are some definitions:

- **Event** — Events are essentially messages produced by an Emitter. Events may or may not carry a data payload. PSR-14 recommends that Events should be immutable. However, in some rare cases, it may be necessary for a Listener to respond back to the Emitter, which may require some additional data. This could be true for a Stoppable Event, where a condition may indicate that no further Listeners will receive the Event. The Event could be updated with the reason for stopping and returning to the Emitter. A Stoppable Event MUST implement the `StoppableEventInterface` and return `TRUE` from `isPropagationStopped()` if that stopping condition has occurred.
- **Listener** — A PHP callable that expects the Event and knows how to react or handle it, either immediately or asynchronously (such as using a queue). Listeners must have only one parameter: the Event for which it's responding. The Listener should typehint against that Event but could typehint against a common interface the Event implements, allowing it to react to variations. Listeners can delegate actions to other code rather than doing all the work, especially if the business logic is located elsewhere. A Listener may enqueue the Event for a secondary process to respond to also. In many modern frameworks, Listeners can be enqueued, supporting asynchronous calls and reducing the load on a server.
- **Emitter** — Calling PHP code that wishes to dispatch an Event. How this is designed or implemented depends on the surrounding code or system. In some frameworks, it's as simple as `emit($myEvent)`, where `$myEvent` is the event object with or without payload.
- **Dispatcher** — This is a service object that is given an Event by an Emitter. It's responsible for passing the Event to any Listeners relevant to the event. According to PSR-14, the Dispatcher should not call the Listeners directly but instead use a Listener Provider to determine the affected Listeners. It is acceptable that the Listener Provider might not return any Listeners. The following rules govern the behavior of the Dispatcher:
  - Must call Listeners synchronously in the order returned by the Listener Provider.
  - Must return the same Event object after invoking the Listeners.
  - Must not return to the Emitter until all Listeners have executed.
  - For a Stoppable Event, the Dispatcher must call `isPropagationStopped()` on the Event before the Listener has been called. If `TRUE`, the Event should be returned to the Emitter and must not call any further Listeners.
- **Listener Provider** — The service object or repository that determines the Listeners for the Event. The provider should simply return the list of Listeners and MUST NOT call them itself. The Listener Provider may not only provide a list of Listeners that take the Event, but it may do so in a specific order or exclude some from the list based on access control. It is also acceptable that a Listener Provider may need to execute lifecycle methods on objects referenced by the Event. How the Listeners are registered is up to the implementer.

Let's use our movie ticket platform for some examples. We have a `TicketPurchasedEvent` (Listing 1) that will

[1] https://www.php-fig.org/psr/psr-14/

[2] https://phpa.me/event-dispatcher

have a payload of purchase and movie data, the TicketPurchasedListener (Listing 2) for our event, an EventDispatcher (Listing 3), and a ListenerProvider (Listing 4).

In our purchase system, our service object has a method like this:

```
protected function purchaseComplete(array $order): void {
    $eventDispatcher = new EventDispatcher();
    //do some other stuff here
    ...

    $event = new TicketPurchasedEvent(
        $order['purchaseData'],
        $order['movieData']
    );
    //this line is our emitter
    $eventDispatcher->dispatch($event);
}
```

When the user has completed their purchase, the purchaseComplete method is called, passing an array of the purchase and movie data. After some other handling, the method finishes by instantiating a new TicketPurchasedEvent and passes it to the EventDispatcher::dispatch() method. The EventDispatcher gets the list of Listeners registered for the Event and begins to iterate through them, instantiating the listener and passing the event to the handle method. Note that I chose to use handle as the name of the method, but it could be anything that makes sense to the implementer. What's important is that the listeners consistently have the same callable method, so the dispatcher knows what to call in the

Listing 1.

```
class TicketPurchasedEvent {
    public function __construct(
        protected array $purchaseData,
        protected array $movieData)
    { }

    public function getPurchaseData(): array {
        return $this->purchaseData;
    }

    public function getMovieData(): array {
        return $this->movieData;
    }
}
```

Listing 2.

```
class TicketPurchasedListener
{
  public function __construct(
    protected TicketPurchasedEvent $event
  ) { }

  public function handle(): TicketPurchasedEvent
  {
    $purchase = $this->event->getPurchaseData();
    $this->ticketBookkeeping($purchase);

    $movie_data = $this->event->getMovieData();
    $this->notifyTheater($movie_data);
  }

  protected function ticketBookKeeping(
    array $purchaseData
  ): void {
    //Does some bookkeeping
    //...
  }

  protected function notifyTheater(
    array $movieData
  ): void {
    //Notifies the theater of the ticket purchase
    //...
  }

  protected function notifyCustomer(
    array $purchaseData,
    array $movieData
  ) {
    //Send purchase confirmation and QR code
  }
}
```

Listing 3.

```
class EventDispatcher implements EventDispatcherInterface
{
  public function dispatch(object $event)
  {
    $provider = $this->getListenerProvider();
    $listeners=$provider->getListenersForEvent($event);
    foreach ($listeners as $listenerClass) {
      if (
        method_exists($event, 'isPropagationStopped')
        && $event->isPropagationStopped()
      ) {
        break;
      }

      $listener = new $listenerClass($event);
      $listener->handle();
    }
  }

  protected function getListenerProvider(
  ): ListenerProviderInterface {
    return new ListenerProvider();
  }
}
```

Listing 4.

```php
class ListenerProvider
             implements ListenerProviderInterface
{
  protected array $listeners = [
    TicketPurchasedEvent::class => [
      TicketPurchasedListener::class
    ],
  ];

  public function getListenersForEvent(
    object $event
  ): iterable {
   if (!array_key_exists($event::class, $this->listeners))
    {
      return [];
    }

    return $this->listeners[$event::class];
  }
}
```

Listing 5.

```php
class PromotionalGiftListener implements Queueable
{
  public function __construct(
    protected TicketPurchasedEvent $event
  ) {
  }

  public function handle(): TicketPurchasedEvent
  {
    //hit the vendor API and register
    //the customer for their gift
  }
}

//Register the new Listener in our ListenerProvider
protected array $listeners = [
  TicketPurchasedEvent::class => [
    TicketPurchasedListener::class,
    PromotionalGiftListener::class, // <-- new listener
  ],
];

//In the EventDispatcher we add a new method
//that will enqueue the listener
public dispatch(object $event) {
  ...

  $listener = new $listenerClass($event);
  $implements = class_implements($event);
  in_array(Queueable::class, $implements)
    ? $this->queueListener($listener)
    : $listener->handle();
}

protected function queueListener(object $listener):void
{
  //queue the listener
}
```

`dispatch` method. The listener then passes the event data to a couple of internal methods that will make some other calls or even emit their own events to process the event.

Now let's assume that we need a new feature where a particular movie, due to some marketing, will give a promotional gift to each attendee. We could add a new method to our `TicketPurchasedListener` but have decided that perhaps that feature is better suited by a `PromotionalGiftListener` that we could send to a queue since an API will need to get hit to register the gift. We don't want our own system held up waiting for a response. So add the following: (See listing 5)

Now, when our `TicketPurchasedEvent` is dispatched, the dispatcher retrieves the listeners, which includes the new `PromotionalGiftListener`. As it iterates and instantiates the listener, it checks if it should be queued instead of handled immediately. In this case, we used a `Queueable` interface to indicate that the listener should be added to the queue by way of the `EventDispatcher::queueListener()` method. We had to change our dispatcher to support queuing, but all we really had to do was create and register a new listener.

Conclusion

Events and listeners are a powerful way to decouple different parts of an application, making it more flexible and extensible. By allowing different systems and objects to listen and react to specific events, complex interactions can be broken down into smaller, more manageable pieces. If you're new to event dispatching, the PHP PSR-14 Event Dispatcher is a great starting point for learning about this important design pattern. Even if you're working with legacy code, there are framework-agnostic libraries available that can help you implement event-driven functionality. So whether you're building a brand new application or trying to modernize an existing one, the benefits of using events and listeners are hard to ignore.

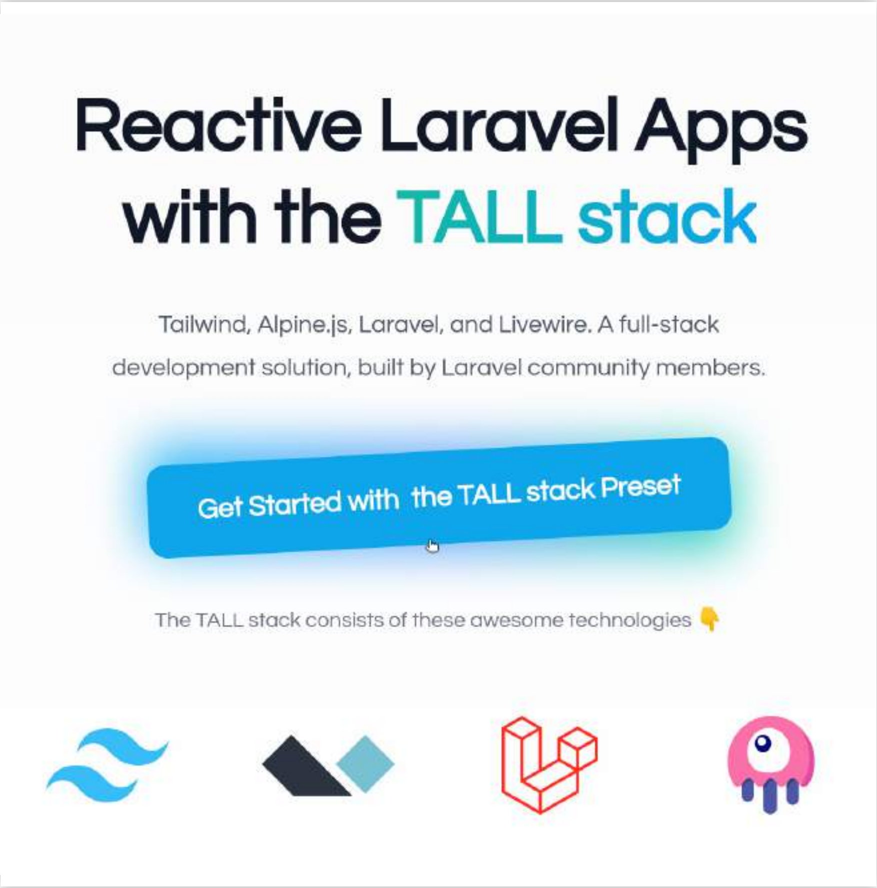
Frank Wallen is a PHP developer, musician, and tabletop gaming geek (really a gamer geek in general). He is also the proud father of two amazing young men and grandfather to two beautiful children who light up his life. He is from Southern and Baja California and dreams of having his own Rancherito where he can live with his family, have a dog, and, of course, a cat. [@frank_wallen](https://twitter.com/frank_wallen)

Standing Tall with the Laravel TALL Stack

Matt Lantz

The TALL Stack is a collection of frameworks for building interactive applications with Laravel. Let's take a closer look at the TALL stack—T(Tailwindcss)—A(Alpine.js)—L(Laravel)—L (Livewire).

The TALL Stack is a collection of frameworks for building interactive applications with Laravel. The group of frameworks handles various elements of modern application development, including front-end and back-end solutions that are built by members of the Laravel community. The combination of frameworks enables developers to focus on PHP-oriented code more than swapping between complex JavaScript frameworks and back to elaborate PHP architectures. The TALL stack consists of four main elements: TailwindCSS, Alpine.js, Laravel, and Laravel Livewire. The collection of tools is intended to help Multi-Page Applications (MPA) resemble Single-Page Applications (SPA), all while avoiding spending too much time in JavaScript nuances and constantly changing frameworks.



Reactive Laravel Apps with the TALL stack

Tailwind, Alpine.js, Laravel, and Livewire. A full-stack development solution, built by Laravel community members.

The TALL stack consists of these awesome technologies 👇

Tailwindcss and Laravel

Laravel is, as many know, a full featured PHP based framework for building modern web applications. It is designed around a simple monolith architecture with a robust queueing system, a powerful templating engine, and a constantly growing community of contributors. Laravel works in a standard PHP framework, with single requests from a URL being processed and the output being sent off. It's a well designed framework for handling MPA applications. To use Laravel with a SPA, you're likely exploring using Laravel exclusively as an API or perhaps simulating a SPA with something like the VITE Stack or TALL Stack. TailwindCSS has become, for much of the Laravel community, the standard CSS framework for building their applications. Tailwind is a utility-first framework which means that their goal is to allow you to build complex components from a constrained set of primitive utilities. Its declarative nature means that rather than having a collection of SCSS files or CSS files that you have to organize and standardize, you can easily write your CSS directly in the class attribute of your HTML elements. From there, TailwindCSS parses your HTML and produces exactly the CSS you need, thereby reducing style bloat and knowledge gaps in CSS frameworks.

Laravel Livewire

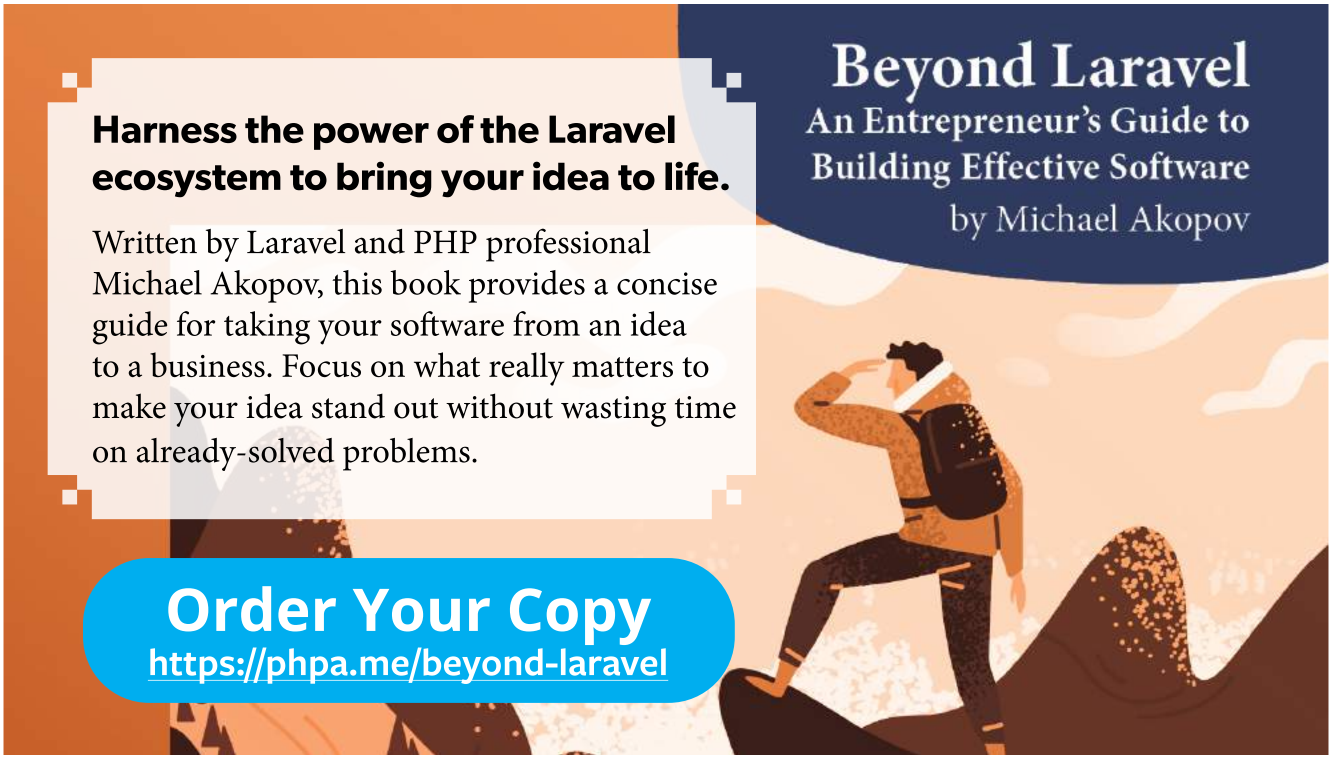
For developers who have worked with Laravel before, Laravel Livewire is much like building Blade components but with more interactivity. It can feel strange to write method calls as JavaScript and their corresponding actions in PHP; the result feels remarkable and refreshing. Though its being SEO friendly is widely boasted, the overall simplicity of single Component files is what sells Livewire to developers. However, there are some things to be cautious about regarding performance. Constant polling of the server can be cumbersome and is not ideal in most cases, but Livewire's events can help reduce some amount of polling. There can also occasionally be issues with how bindings work when mixing native JavaScript and DOM content changes from the loaded diff via Livewire's rendering. However, as mentioned before, it can be worked with elegantly using Livewire's event listeners and well-structured front-end code. Laravel Livewire's simplicity in action binding and its complete removal of the hard JavaScript parts lend itself well to a community of back-end developers who often encounter cases where basic MPA experiences aren't enough for modern web application users. When exploring whether or not to use a full-featured JavaScript framework for a few interactive experiences vs. implementing what are essentially blade components with seamless loading, leaning into the TALL Stack is likely the most efficient path forward.

Alpine.js

Question: Is a lightweight javascript library that, by its own terms, is like jQuery, good for the modern web? It's the perfect Javascript accomplice to Laravel Livewire. When used together, Alpine and Livewire can be orchestrated to make fewer full server-roundtrip requests, which will make your page feel more responsive at times. Alpine.js was built to provide a Vue-like experience without all the overhead of Vue and the complexity of a full-featured JavaScript framework. It works directly in the DOM rather than creating a virtual DOM. Much like how TailwindCSS is engineered to create the CSS needed by parsing through the HTML, Alpine.js enables actions based on the declarations in your HTML. That can significantly help reduce maintenance complexities where you have to sift through multiple JS files and dependencies to fix that one component on your application. Overall, Alpine.js can handle most interactive use cases regarding content in a web application while removing all the bloat and knowledge gaps that can come with full-featured frameworks and their corresponding dependencies.

The TALL stack gained most of its popularity because it's focused on simple JavaScript with PHP-centered components for interactive elements with backend storage. It's use of TailwindCSS and Laravel mostly revolves around the fact that it's built within the Laravel community, and those are the standard frameworks for styles and back-end structures. Its creative use of simple declarative JavaScript injections in HTML allows for easy-to-read code in fewer places and fewer build steps required for testing components. Alpine.js and Laravel Livewire were both created by Caleb Porzio, who has also expanded Alpine.js with a set of UI components. The TALL stack provides a fantastic pathway to building applications that do not rely on full-featured JavaScript frameworks and reduce the knowledge base required to maintain and upgrade applications. Because of this, it's clear that the TALL stack will stick around for quite some time as a fantastic toolset for building efficient applications for the modern web.

Matt has been developing software for over 13 years. He started his career as a developer working for a small marketing firm, but has since worked for a few Fortune 500 companies, lead a couple teams of developers and is currently working as a Cloud Architect. He's contributed to the open source community on projects such as Cordova and Laravel. He also made numerous packages and helped maintain a few. He's worked with Start Ups and sub-teams of big teams within large divisions of companies. He's a passionate developer who often fills his weekend with extra freelance projects, and code experiments. @Mattylantz



Self-worth

Beth Tucker Long

As the new year dawns, the snow is flying (at least where I am), and so are the resolutions. I can't even count how many emails I have gotten from companies about how they can help me stick to "my resolution" to lose weight, exercise more, read more books, learn a new language…the list goes on and on.

I normally don't participate in new year's resolutions, but this year deserves a commitment of some kind. Wait, no. Not the year; I deserve a commitment, and so do you—a commitment to improving your mental health and encouraging those around you to do the same.

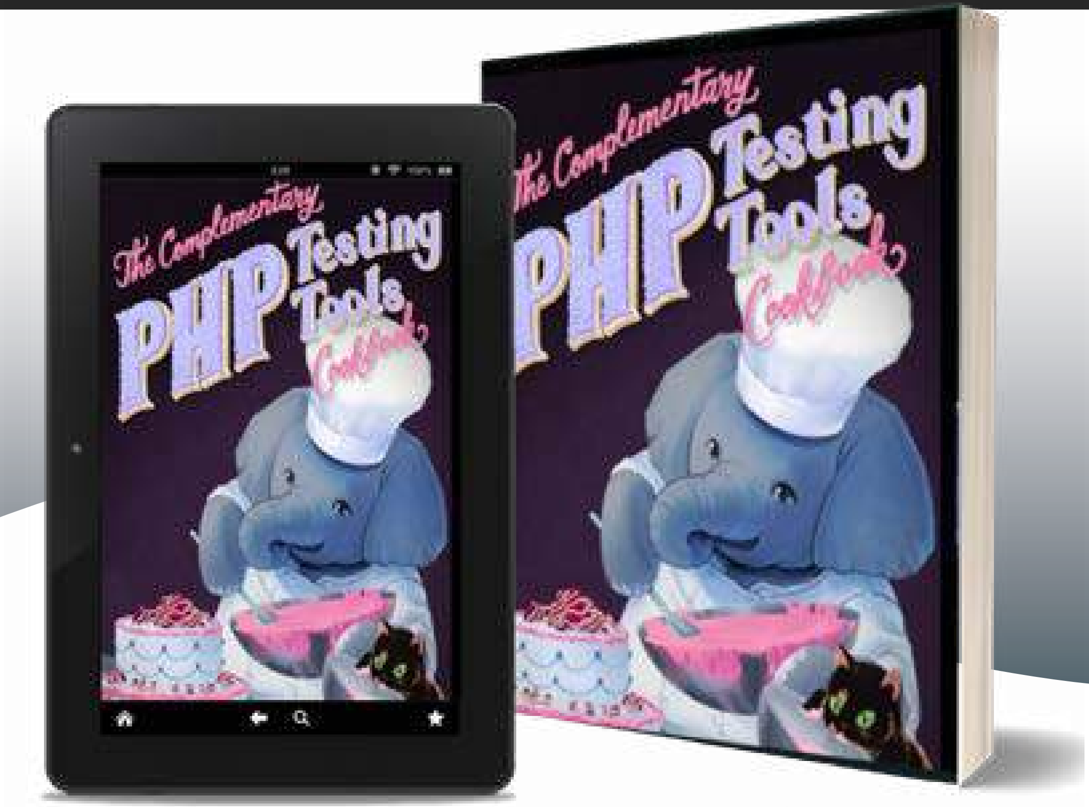
I have spent the last year learning a lot about mental health and about myself. One of the things I have learned is that I love doing things for others. It makes me so happy to teach someone something new or to organize an event that people will enjoy attending. I also learned that I had wrapped too much of my self-worth into the things I do for other people. It's good to do things for other people, but it is very important to know that self-worth is more than how happy you can make other people. I often forget this, so this year I am making a commitment to remember that I am a worthwhile person who chooses to do things; I am not a worthwhile person because of the things I do.

I hope this is something you know too. It's great if you do things for other people, but you are still a valuable human being, even if you do nothing for other people. If this is something you struggle with, too, I am here to tell you: You are more than the things you do.

As I have learned to recognize this in myself, I have also begun to recognize it in others. Do you know someone who is always ready to volunteer to help? Someone who is always working on something to help others? Maybe it's a user group leader or a club president. Perhaps it's the person at work who always organizes learning events or social outings, even though that's not part of the job. Maybe it's the friend who drops everything at a moment's notice whenever you need something. There are so many people in our lives who challenge us to be better, provide opportunities for us to grow, and support us when we are low. Make sure they know that you value them for more than the things they do for you. Make sure they know that you see them as a person, not as a service delivery system. Make sure everyone, including you, knows: You are not defined by your productivity.

I wish each and every one of you a year full of learning, laughter, and good mental health care. You are worth it.

Beth Tucker Long is a developer and owner at Treeline Design, a web development company, and runs Exploricon, a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development and Full Stack Madison user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @e3BethT


The Complementary
PHP Testing Tools
Cookbook