



Another Bright Idea



Illuminating Smart Light Bulbs with PHP

Building Solid and Maintainable PHP Apps

ALSO INSIDE

Education Station:

Event-Driven
Programming

PHP Puzzles:

Clues for Hues

Security Corner:

Assessing
Cybersecurity Risks

DDD Alley:

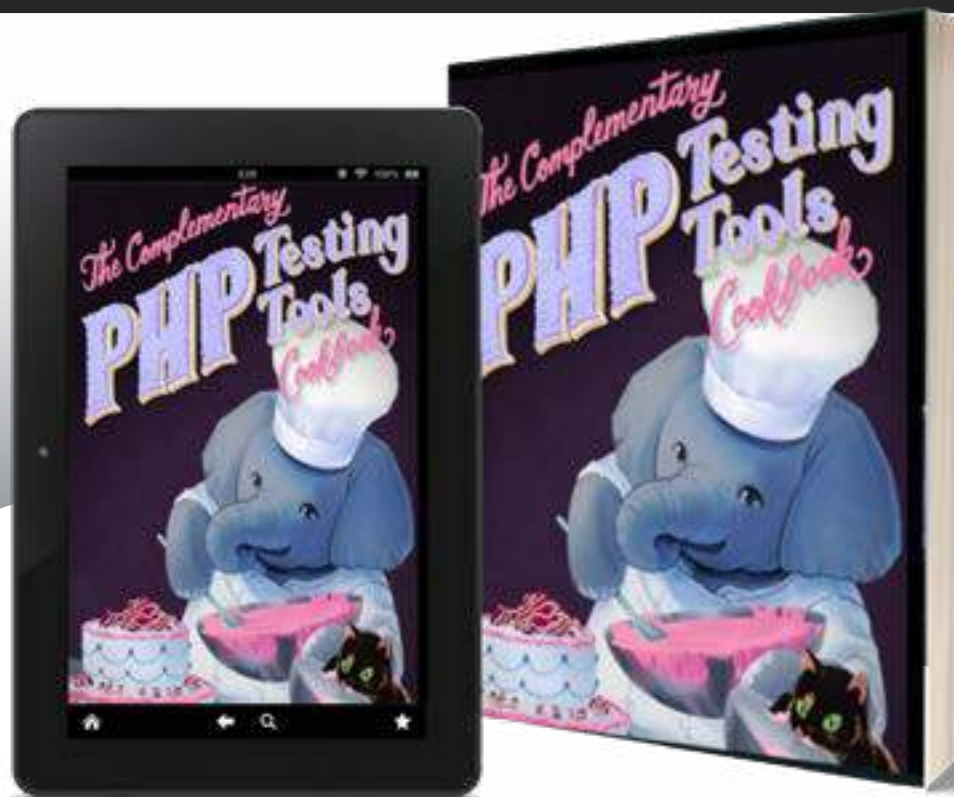
Random and Rare
Failures

The Workshop:

A Night With
Symfony

finally{ }:

Pulled from All
Angles



Learn how a Grumpy Programmer approaches improving his own codebase, including all of the tools used and why.

The Complementary PHP Testing Tools Cookbook is Chris Hartjes' way to try and provide additional tools to PHP programmers who already have experience writing tests but want to improve. He believes that by learning the skills (both technical and core) surrounding testing you will be able to write tests using almost any testing framework and almost any PHP application.

Available in Print+Digital and Digital Editions.

Order Your Copy
[**phpa.me/grumpy-cookbook**](http://phpa.me/grumpy-cookbook)

CONTENTS

JUNE 2022

Volume 21 - Issue 6



2 Another Bright Idea

Eric Van Johnson

3 Illuminating Smart Light Bulbs With PHP

Sherri Wheeler

7 Building Solid and Maintainable Php Applications

Dariusz Gafka

21 Event-Driven Programming

Education Station

Chris Tankersley

25 Random and Rare Failures

DDD Alley

Edward Barnard

31 Assessing Cybersecurity Risks

Security Corner

Eric Mann

33 A Night With Symfony

The Workshop

Joe Ferguson

40 Clues for Hues

PHP Puzzles

Oscar Merida

44 Pulled From All Angles

finally{}

Beth Tucker Long

Edited with Light Bulb Moments

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

Copyright © 2022—PHP Architect, LLC
All Rights Reserved

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.

php[architect], php[a], the php[architect] logo, PHP Architect, LLC and the PHP Architect, LLC logo are trademarks of PHP Architect, LLC.

Another Bright Idea

Eric Van Johnson

One of the best feelings you can have is a “lightbulb moment.” Whether it’s a great idea or clarity of understanding on a problem you are working on, coding practice, architecture, design pattern, or some other paradigm, that moment when you can connect the dots and make sense of something that moments earlier seemed utterly confusing.

Fun fact, the comic strip *Felix the Cat*¹ is credited as the first comic strip to use the lightbulb to symbolize conceiving an idea. If you’ve been a developer long enough, this has happened to you, even with an established language like PHP that continues to mature.

Let me shine some light on what is coming to you in this month’s magazine.

If you enjoy coding in PHP and have a Philips Hue smart bulb, this month’s feature article, “Illuminating Smart Light Bulbs with PHP,” from Sherri Wheeler, is for you. She shares a great package and code on how to control your lights using PHP. Dariusz Gafka’s contribution “Building Solid and Maintainable PHP Applications Using DDD and Messaging with Ecotone Framework,” touches on Domain-Driven Design and the Ecotone Framework with tons of sample code.

In Education Station this month, Chris Tankersley introduces us to the often-overlooked Programming Paradigm of Event-Driven Programming in his article “Event-Driven Programming.” He opens the door to the concept of your application listening for events and then acting on those events when they happen. Eric Mann brings us “Assessing Cybersecurity Risks” in this month’s Security Corner. Understanding different threat categories are essential to any developer who plans to put a line of code on the public internet.

Eric breaks this down and helps you identify and grade risk.

Now that Edward Barnard has gotten you started and organized with his column from last month, this month, he will address failures in his article “Random and Rare Failures.” He will explain why you want to be on the lookout for random and rare failures and how you can capture them.

If you’ve ever been curious to give the Symfony Framework a try, I’ve got some excellent news. In this month’s The Workshop, Joe Ferguson brings you “A Night With Symfony.” He shares everything you’ll need to know to get you up and running with the Symfony framework, from installing it, getting a server up and running, and writing code.

Time to dust off your brain cells for Oscar Merida’s PHP Puzzles column “Clues for Hues,” where he shows solutions for last month’s Guess the Colors puzzle. Hey, Hue smart lights can change color. I bet you could develop a crossover article after reading this month’s magazine. And as always, he brings another challenge for you to work on until the next release.

And in Finally{}, Beth Tucker Long shares “Pulled From All Angles” where she points out some of the hypocrisy in our industry and some of the back and forth of the “shoulds” and “should nots.”

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 announcements. Make sure you're connected with us.

- Subscribe to our list: <http://phpa.me/sub-to-updates>
- Twitter: [@phparch](https://twitter.com/phparch)
- Facebook: <http://facebook.com/phparch>

Download the Code Archive:

http://phpa.me/June2022_code

¹ *Felix the Cat*: <https://grammarist.com/?p=34468>

Illuminating Smart Light Bulbs With PHP

Sherri Wheeler

PHP is becoming more popular for controlling Internet Of Things devices like home automation, cameras, doorbells, and even light bulbs. In this article, we'll shed some literal and figurative light on controlling Philips Hue smart light bulbs using an existing PHP library.

Philips sells a popular line of lighting devices in white and color variations in a variety of bulb formats. I personally have the colored bulbs which we use in the living room to give some color-themed ambiance to our TV watching. We also have a pair that we use on our nightstands to create a gradual and gentle wake-up experience. There is an official app that can be used to control the bulbs and create various routines and schedules. There are also several unofficial apps that provide similar functionality. However, what if we get the bright idea to do something very specific or even use light as an indicator or output for an application. Imagine changing a light's color based on a status value: like turning it to red if a task is overdue!

You might not automatically reach for PHP to do this kind of work. When I first looked into it, I wasn't expecting to find a complete Composer package that does this- but it exists! The original **Phue** GitHub project was [sqmk/Phue](https://github.com/sqmk/Phue)¹, which was forked and updated by [neoteknic/Phue](https://github.com/neoteknic/Phue)². We'll be using my own fork of this package, and we'll go through the basics of connecting to and using the Hue bulbs.

Find the Bridge

First, you'll want to add my fork³ to your project:

```
composer require syntaxseed/phue
```

Note: As of the time of this writing, I have a few outstanding PRs to the original project [neoteknic/Phue](https://github.com/neoteknic/Phue)—some bug fixes and a new feature. If the PRs have been merged by the time you read this, you might want to use the package [neoteknic/Phue](https://github.com/neoteknic/Phue) because if they have merged them, I don't plan to maintain my fork for the long term.

Don't forget to require Composer's autoload script⁴ into your application's bootstrapping.

The absolute first thing we need to do is find the IP address of the bridge (the device which connects to your home router and controls the bulbs) and create a user on it so that we have permission to control the lights. When we give commands to

the bulbs, what we are actually doing is communicating to the bridge which controls the bulbs.

The library has a helpful script which you can call from within the `vendor/neoteknic/Phue` directory:

Listing 1.

```
1. ./bin/phue-bridge-finder
2.
3. # Output:
4.
5. Philips Hue Bridge Finder
6.
7. Checking meethue.com if the bridge has phoned home:
8.     Request succeeded
9.
10. Number of bridges found: 1
11.     Bridge #1
12.         ID: 00123123123ea598f6
13.         Internal IP Address: 192.168.1.2
```

That internal IP address is what you'll use for connecting to your bridge, so make note of it.

If the script doesn't work (for example, if your bridge hasn't phoned home), there are other ways to find your bridge's IP described in the project's wiki⁵. Using `nmap` works for me as well:

```
# Replace 10.0.1.255 with your network broadcast IP
nmap -sP 10.0.1.255/24 > /dev/null;
sudo arp -na | grep "at 00:17:88"
```

Output:

```
? (192.168.1.2) at 00:17:88:a5:98:f6 [ether] on enp5s0
```

Brilliant! We now have an IP address to communicate with the bridge.

Creating a User On the Bridge

It would be very problematic if just anyone on your network could send commands to the bridge. To prevent this, you must authorize a user on the bridge by sending a request while pressing the physical button on the top of the bridge.

1 [sqmk/Phue](https://github.com/sqmk/Phue): <https://github.com/sqmk/Phue>

2 [neoteknic/Phue](https://github.com/neoteknic/Phue): <https://github.com/neoteknic/Phue>

3 [fork](https://github.com/syntaxseed/phue): <https://github.com/syntaxseed/phue>

4 [autoload script](https://getcomposer.org): <https://getcomposer.org>

5 [project's wiki](https://phpa.me/github-neoteknic-phue-wiki-bridge): <https://phpa.me/github-neoteknic-phue-wiki-bridge>

Again we have a nice helpful script that will do this for us. Run this command along with the bridge IP that you just found, and while you see the “Waiting...” message, press the big button on top of your bridge.

```
./bin/phue-create-user 192.168.1.2
```

Output:

```
Testing connection to bridge at 192.168.1.2
Attempting to create user:
Press the Bridge's button!
Waiting.....
```

If the command succeeds, you’ll receive a username that looks like a series of letters and numbers. This username and the IP address of the bridge are what you’ll need for the rest of this project, so keep a record of them.

Listing Lights and Light Groups

Now for the good stuff! Hue allows you to group your lights into ‘rooms.’ This way, you can control multiple bulbs simultaneously by setting them all on, off, or applying a ‘scene’ which sets the lights in a room to various colors. For example, I have a ‘superman’ scene for watching the TV series Superman and Lois, which sets one light to blue and one light to red.

Before we can control lights or groups we first need to list them. We can use the Phue package to do just this. First we create a Client object using the bridge IP and the username.

Listing 2.

```
1. use \\Phue\\Client;
2. use \\Phue\\Command\\Ping;
3. use \\Phue\\Command\\GetLights;
4. use \\Phue\\Command\\SetGroupState;
5. use \\Phue\\Command\\SetLightState;
6. use \\Phue\\Command\\CreateScene;
7. use \\Phue\\Command\\SetSceneLightState;
8. use Phue\\Transport\\Exception\\ConnectionException;
9.
10. require_once '../vendor/autoload.php';
11.
12. $client = new Client('192.168.1.2', 'abcdefg123456');
```

Next, in Listing 3, we’ll send a test ping command to the bridge, and if that doesn’t throw an exception, we’ll request and display the list of lights and the list of groups. I’m testing my code in a browser, so I’ll use some HTML to improve the formatting.

This code will give us the IDs of individual lights and light groups. We’ll use these IDs to send commands to affect those specific items.

It’s important to note that the `getLights()` and `getGroups()` functions return arrays indexed with the IDs of the elements.

Listing 3.

```
1. try {
2.     $client->sendCommand(
3.         new Ping
4.     );
5.
6.     $lights = $client->getLights();
7.     $groups = $client->getGroups();
8.
9.     foreach ($lights as $lightId => $light) {
10.        echo "Id #{$lightId} - {$light->getName()}<br>";
11.    }
12.
13.    echo('<hr>');
14.
15.    foreach ($groups as $group) {
16.        $groupLightIds = implode(' ', $group->getLightIds());
17.        echo <<<EOT
18.            Id #{$group->getId()} - {$group->getName()}
19.            (Type: {$group->getType()})
20.            Lights: {$groupLightIds}<br>
21.        EOT;
22.    }
23.
24. } catch (ConnectionException $e) {
25.     echo 'ERROR: There was a problem accessing Hue bridge.';
26. }
```

Turning Lights On and Off

It’s time to banish the darkness! The most basic thing you can do with your lights is to turn them on and off. To do so, we first get the list of light objects and use one of them to set its on state. Hue bulbs don’t flick to an instantaneous, fully on or off state. Instead, you will see them very quickly grow in brightness from off to maximum and vice versa. You can tweak this with transition time- more on that later.

When you turn the lights on and off programmatically for the first time, you’ll notice they will be at their last color and brightness setting. So if you don’t see anything happening, double-check with the official app that you don’t have the brightness set way down, or just peek at the next section.

```
$lights = $client->getLights();
```

```
$light = $lights[1];
```

```
$light->setOn(true);
sleep(2); // Wait for 2 seconds.
$light->setOn(false);
```

Color, Brightness, and Effects

Let’s get more interesting. If you have the color version of the lights, you can change them to any color on the RGB spectrum. You can also set color bulbs and white bulbs to different

Listing 4.

```

1. $lights = $client->getLights();
2.
3. $light = $lights[1];
4.
5. $light->setOn(true);
6. $light->setBrightness(255);
7.
8. $light->setRGB(255, 0, 0); // Red
9. sleep(2); // Wait for 2 seconds.
10. $light->setRGB(0, 255, 0); // Green
11. sleep(2); // Wait for 2 seconds.
12. $light->setRGB(0, 0, 255); // Blue

```

brightness levels. The Phue library has built-in functions for these settings as shown in Listing 4.

The `setBrightness` function takes a value from 0 to 255 to scale the brightness from off to maximum. There are several ways to change the color including `setHue`, `setSaturation`, and `setXY`. However, I find the easiest function for me is `setRGB` (which many of us web developers are familiar with) which takes red, green, and blue values from 0 to 255.

Another method to change the color is using `setColorTemp`, which takes an integer value between 153 (6500K) to 500 (2000K). This function sets the light along a temperature range from warm and yellow to white to cool blue light (similar to outdoors).

There is also a `setEffect` function which can be set to either `none` or `colorloop`. The `colorloop` effect will cycle smoothly through all the hues using the current settings for brightness and saturation.

Listing 5.

```

$lights = $client->getLights();

$light = $lights[1];

$light->setOn(true);
$light->setBrightness(255);

$light->setEffect('colorloop');

```

Creating a Candle Effect

Let's build on what we've learned so far and create a script which will simulate a flickering candle effect. We'll bundle multiple settings into one command so we can execute them all at once. The `SetLightState` command object takes a light ID or a `Light` object in the constructor, and we can chain methods off this object to build the command.

Our candle effect involves changing brightness and temperature for a variety of durations to have the bulb change

in a random but pleasant way. We don't want to vary the brightness or temperature too wildly, so we'll choose conservative ranges.

While testing, we'll loop a small number of times. Later we might put this in an infinite loop inside a long running process. There is also a way to create long running effects which are sent to the bridge all at once and will keep playing until another command is sent, but that is beyond the scope of this article.

Listing 6.

```

1. // Flickering candle effect.
2. for ($i = 1; $i < 100; $i++) {
3.     // Randomly choose values.
4.     $brightness = rand(20, 50);
5.     $colorTemp = rand(420, 450);
6.     $transitionTime = rand(0, 3) / 10;
7.     $waitTime = $transitionTime;
8.
9.     // Setup command.
10.    $command = new SetLightState(1);
11.    $command->brightness($brightness)
12.        ->colorTemp($colorTemp)
13.        ->transitionTime($transitionTime);
14.
15.    // Send command.
16.    $client->sendCommand($command);
17.
18.    // Sleep for transition time plus extra for request time.
19.    usleep($waitTime * 1000000 + 25000);
20. }

```

Inside the loop, we choose a random brightness, temperature, and transition time multiplier. Transition time is how long it will take the bulb to go from its current state to the new state defined in the command. The value is a multiple of 100ms, so a transition time of 10 will be one second. We'll set it to some fraction out of 10, so we are always somewhere between 0 and 30ms.

The time to sleep between loop intervals will be the same as the transition time to create quick flickers and slower burns.

We create a new `SetLightState` command object with the ID of the bulb we're using. Then we chain methods to set the brightness, color temperature, and the transition time from the randomized values we generated. We then use the `sendCommand` method on the client to send it to the bridge. Finally, we sleep for our wait time and repeat the loop.

Using Scenes

Scenes allow you to set the lights in a group to specific colors as a shortcut or bookmark your favorite settings. Let's create a PHP Architect theme for the two bulbs in my living room.

Listing 7.

```

1. // Set up the scene and its lights.
2. $client->sendCommand(new CreateScene('phparch', 'PHP Architect', [1, 2]));
3.
4. // Configure the state of light ID 1 in this scene.
5. $command = new SetSceneLightState('phparch', 1);
6. $command = $command->brightness(200)
7.     ->rgb(220, 90, 33); // Orange
8. $client->sendCommand($command);
9.
10. // Configure the state of light ID 2 in this scene.
11. $command = new SetSceneLightState('phparch', 2);
12. $command = $command->brightness(200)
13.     ->rgb(60, 70, 175); // Purple
14. $client->sendCommand($command);

```

We start by creating a new scene with an ID, name, and an array of the lights in this scene. We'll name this one "PHP Architect." When we send that scene to the bridge, it will use the current state of the lights in the scene. To change that, we use the `SetSceneLightState` command to choose the scene (by ID) and the light, then set the brightness and color of the light in that scene. I used the orange and purple colors lifted from the PHP Architect website.

Now we can turn on our scene to see it working. A scene is actually applied to a group, so make sure you set up your scene to apply to lights that are in that group. To confirm it's using our scene, we first set the lights to a color that is not in the scene and wait a couple seconds before using the `SetGroupState` command to change to our scene.

Listing 8.

```

1. $lights[1]->setOn(true);
2. $lights[1]->setRGB(0, 255, 0); // Green
3. $lights[2]->setOn(true);
4. $lights[2]->setRGB(0, 255, 0); // Green
5. sleep(2);
6.
7. $command = new SetGroupState($groups[1]);
8. $command->scene('phparch');
9. $client->sendCommand($command);

```

Illuminating! Note that with this library (which uses the Hue API version 1), the scenes are set to have the value of 1 for the recycle flag. This flag means that when the bridge reaches its maximum number of scenes, it will delete the recyclable and unlock the scene that was last used the longest time ago.

When we are done with the scene, we can delete it from the bridge with the `delete` method on the `Scene` class. This was functionality that I added to my fork and is in a PR to the upstream package.

```

$scenes = $client->getScenes();
$scenes['phparch']->delete();

```

Further Reading

The documentation for Phue is in a single readme file with an overview of the library. There is additional functionality that Phue can do that isn't in the readme, but you'll have to dig into the library's source code to find out more. There is also a directory of examples that I found to be very helpful.

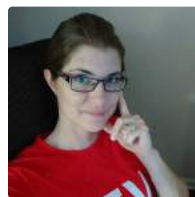
If you're interested in going deeper into what you can do with your Hue lights, Philips has an API reference website⁶. The documentation is not great, and it requires you to create a user account to access the API reference, but it exists. There is also a v2 of the API, which is different from what is used by Phue.

Conclusion

As a PHP developer, I'm very excited to see a growing body of home automation projects that can be built with PHP. We need developers to push the boundaries of PHP's traditional use.

However, the Phue project has not had major updates in years, and the other libraries listed on the Philips developer website for PHP are also quite old. Perhaps you'll be the one to extend Phue or build a new library that takes advantage of modern PHP and the latest Hue APIs!

In this article, we had a broad tour of what the Phue library can do with the Hue API. Imagine using Hue lights to change colors for application indicators or flash when a doorbell is rung, or a motion sensor detects a visitor. Imagine changing the color of a light depending on the temperature of your pool or fish tank. PHP has certainly grown to do more than web development, and libraries like Phue are just the beginning.



Sherri is a web software developer and business professional with over 15 years of experience working with PHP. She currently operates her company Avinus Web Services, and for fun she creates games and open source software. When not at the computer you'll find her camping, swimming and making forts with her husband and three sons in Ontario, Canada. Connect with her on Twitter. [@SyntaxSeed](#)

⁶ API reference website:
<https://developers.meethue.com/develop/get-started-2/>

Building Solid and Maintainable Php Applications Using DDD and Messaging with Ecotone Framework

Dariusz Gafka

Each business has its set of problems that developers need to solve. In DDD, those problems describe how the business works and the user flows—the so-called Problem Space. On the other side, when we want to provide functionality by writing the code to solve those problems, this is Solution Space. DDD is about those two things. We need to have a problem in order to apply a solution.

What is Domain Driven Design?


A series of patterns help us in both of those things. For the Problem Space, we can:

1. Just simply discuss it with the stakeholders/ product-owners
2. Have event storming sessions, which provide us with information about meaningful events and flows in the system
3. We can do context mapping to see how different parts of the business interact with each other

All those things help us better capture the problems we want to solve. On the other hand, the Solution Space contains patterns applied on the code level. For example:

1. Aggregates that provide encapsulation for a given set of behaviors
2. Events that are responsible for informing different parts of the system about something meaningful for the business that just happened
3. Repositories that provide an abstraction layer over the storage system

Those patterns are also called building blocks. Like building a house, we want to have a solid foundation from good quality blocks, so we can feel safe when making changes on the upper floors.

 *Business-related code often works for years. Problem/ Solution Space patterns are here to make us explicit about what we want to solve, so we can extend and adjust the house to our changing needs.*

Preparation for House Repair

There are a lot of materials available about Problem Space. If we talk to business people about features we build, we are already exploring this space. We will focus on Solution Space and the code we write.

Repairing Our House

We will start with some general practices to make our code more maintainable in the long term. These are general guidelines that can be applied to any codebase.

We will build a simple shop and execute it from the command line for simplicity. However, any of this code can be applied for HTTP usage too.

We will build an e-book shop with the following requirements:

- We will be able to register new e-books for a given price.
- Users can order e-books and pay for them with a credit card.
- A user can place an order by simply providing their email address.
- Once payment is successful, we will send an email containing the e-books.
- After 3 successful orders, we will reward the user with a promotion for a 10% decrease for all next orders.

We will be using Ecotone Framework and building a clear domain model. DDD is about having code clear of external tooling, and Ecotone supports this idea. In most cases, you will see that it provides glue code by taking care of technical aspects, so you focus on what matters, which is business-related code.

Ecotone can be run stand-alone in the so-called Lite version, also together with Symfony or Laravel. We will be using the Lite version. Everything that we will apply to the code here can also be applied to the above frameworks.

Making Use of Cms

From a high-level Problem Space perspective, we have 3 options to solve this kind of system.

Listing 1.

```

1. class EbookController
2. {
3.     ...
4.     public function registerEbook(string $requestAsJson): void
5.     {
6.         $data = json_decode($requestAsJson, true,
7.             flags: JSON_THROW_ON_ERROR);
8.         $this->ebookService->registerEbook($data);
9.     }
10.
11.     public function updateEbook(string $requestAsJson): void
12.     {
13.         $data = json_decode($requestAsJson, true,
14.             flags: JSON_THROW_ON_ERROR);
15.         $this->ebookService->updateEbook($data);
16.     }
17.
18.     public function getEbook(string $ebookId): string
19.     {
20.         return json_encode(
21.             $this->ebookService->getEbookById($ebookId));
22.     }
23. }

```

Listing 2.

```

1. class EbookService
2. {
3.     ...
4.     public function registerEbook(array $data): void
5.     {
6.         $this->validateData($data);
7.
8.         $this->connection->insert("ebooks", [
9.             "ebook_id" => $data["ebookId"],
10.            "title" => $data["title"],
11.            "content" => $data["content"],
12.            "price" => $data["price"]
13.        ]);
14.    }
15.
16.    ...
17.    private function validateData(array $data): void
18.    {
19.        if ($data["price"] < 0) {
20.            throw new InvalidArgumentException(
21.                "Ebook price must be higher or equal to 0");
22.        }
23.        if (strlen($data["title"]) <= 0) {
24.            throw new InvalidArgumentException(
25.                "Title must contain any words");
26.        }
27.        if (strlen($data["content"]) < 10) {
28.            throw new InvalidArgumentException(
29.                "Content must be at least 10 characters long");
30.        }
31.    }
32.    ...


```

1. We could potentially use a CMS to solve this problem (requirements) which is a quick way of going to market without building everything from scratch. CMSs have tons of plugins that can be helpful to fulfill specific business needs.
2. If we are planning long-term support and want to have an advantage over our competition, we would like to adjust quickly to the market and stay open to new initiatives and ideas. In that case, building your own application makes a lot of sense, as it's much easier to change your own platform compared to a 3rd party platform. You may then extend it in a way you would like and react much quicker to market changes.
3. You may also go for a hybrid solution, where part of the system will be powered with 3rd party software, and the business core will be written internally. Doing so is often the safest option, as we can focus on what brings in the money by extending it in a way we want and make use of 3rd party software for the rest.

Solution Space patterns apply to the code we own, as we need a lot of freedom in the way we structure the code. As we want to try DDD and be flexible and extensible, our Shopping system applies to the second option.

We will build three versions, where each one will be derived from refactoring the previous one. Thanks to this process, you can see how to refactor code to make it more readable and solid. Our first version will be an SQL-based version.

1st Version—Sql All the Things

 You can find all the code we will be working on in my [github repo](https://github.com/dgafka/ebook-shop)¹.

Let's examine our first version of the app, based on SQL. We require the following packages: doctrine/dbal ecotone/lite-dbal-starter

Let's start by registering an Ebook. For this, the following business rules apply:

- the price must be higher or equal to 0
- the e-book name must not empty
- the e-book content must have at least 10 characters

We have an EbookController (Listing 1), responsible for handling registration, updating, and fetching a single ebook. We have also covered business rules in the validateData() method within the EbookService in Listing 2. In DDD, that kind of protective rule we call invariants.

Now that we can register e-books, we need to allow users to order them using a credit card, and then we will send them through email.

¹ repo: <https://phpa.me/dgafka-ebook-shop>

For this, the following business rules apply:

- The credit card must be correct to avoid a scam transaction, as we are paying fees to the Payment Gateway no matter if the transaction was successful or not.
- We must perform the payment before we send ebooks over email. We want to be sure that the actual payment has been made before sending anything.
- Email must be valid. We want to be sure that the user did not make a typo before we charge him.

We start this process with our `OrderController` and our `OrderService` (Listing 4).

1. Our `placeOrder` method validates emails and credit cards and performs steps related to placing the order.
2. Stores order for auditing purposes in the database
3. Stores the current state of promotion in the database in the `PromotionService` (Listing 5).
4. Makes an external HTTP call to Payment Gateway to charge the client for e-books. For simplicity reasons, we are faking this call in the `PaymentGateway`.
5. Sends an email using an external Email Provider. For simplicity reasons, we are faking this call in the `EmailService`.

Listing 4.

```

1. class OrderService
    ...
11. // 1
12. public function placeOrder(array $data): void
13. {
14.     if (!filter_var($data["email"], FILTER_VALIDATE_EMAIL)) {
15.         throw new \InvalidArgumentException("Email is incorrect: " . $data["email"]);
16.     }
17.     if (!($data["creditCard"]["validTillMonth"] >= 1
18.         && $data["creditCard"]["validTillMonth"] <= 12)) {
19.         throw new \InvalidArgumentException(
20.             "Month validity must between 1-12, got: "
21.             . $data["creditCard"]["validTillMonth"]);
22.     }
23.     if (strlen($data["creditCard"]["validTillYear"]) !== 4) {
24.         throw new \InvalidArgumentException("Year must have 4 characters");
25.     }
26.     if (strlen($data["creditCard"]["cvc"]) !== 3) {
27.         throw new \InvalidArgumentException("Cvc code must be contain 3 characters");
28.     }
29.     if (!$this->validateLuhn($data["creditCard"]["number"])) {
30.         throw new \InvalidArgumentException("Credit card number must be valid");
31.     }
32.     $relatedEbooks = [];
33.     foreach ($data["ebookIds"] as $ebookId) {
34.         $relatedEbooks[] = $this->ebookService->getEbookById($ebookId);
35.     }
36.
37.     $price = 0;
38.     foreach ($relatedEbooks as $ebook) {
39.         $price += $ebook["price"];
40.     }
41.     $price = $this->promotionService
42.         ->isGrantedToPromotion($data["email"]) ? ($price * 0.9) : $price;
43.     $this->connection->beginTransaction();
44.     try {
45.         // 2
46.         $this->saveOrder($data, $price);
47.         // 3
48.         $this->promotionService->increaseOrderAmount($data["email"]);
49.         // 4
50.         $this->paymentGateway->performPayment($data["creditCard"], $price);
51.         // 5
52.         $this->emailService->sendTo($data["email"], $relatedEbooks);
53.
54.         $this->connection->commit();
55.     } catch (\Throwable $exception) {
56.         $this->connection->rollback();
57.         throw $exception;
58.     }
59. }

```

Listing 5.

```

1. class PromotionService
2. {
3.     public function __construct(private Connection $connection) { }
4.
5.     public function isGrantedToPromotion(string $emailAddress): bool {
6.         if ($promotion = $this->getCurrentPromotion($emailAddress)) {
7.             return $promotion["amount_of_orders"] >= 3;
8.         }
9.         return false;
10.    }
11.
12.    public function increaseOrderAmount(string $emailAddress): void {
13.        if ($promotion = $this->getCurrentPromotion($emailAddress)) {
14.            $this->connection->update("promotions", [
15.                "amount_of_orders" => $promotion["amount_of_orders"] + 1
16.            ], ["email" => $emailAddress]);
17.            return;
18.        }
19.
20.        $this->connection->insert("promotions",
21.            ["email" => $emailAddress, "amount_of_orders" => 1]);
22.    }
23.
24.    private function getCurrentPromotion(string $emailAddress): ?array {
25.        $result = $this->connection->executeQuery(<<<SQL
26.            SELECT * FROM promotions WHERE email = :emailAddress
27.            SQL, ["emailAddress" => $emailAddress])->fetchAssociative();
28.        return $result ?: null;
29.    }
30. }

```

Listing 9.

```

1. $dogStoryEbookId = 1;
2. $ebookController->registerEbook(json_encode([
3.     "ebookId" => $dogStoryEbookId,
4.     "title" => "Happy Dog Story",
5.     "content" => "Dog went for work a walk and is happy!",
6.     "price" => 10
7. ]));
8. $cookbookId = 2;
9. $ebookController->registerEbook(json_encode([
10.    "ebookId" => $cookbookId,
11.    "title" => "Cookbook - Home Recipes",
12.    "content" => "To make scrambled eggs, you need to first have eggs.",
13.    "price" => 20
14. ]));
15.
16. echo "Making order for two books\n";
17.
18. $orderController->placeOrder(json_encode([
19.    "email" => "johnybravo@o3.en",
20.    "ebookIds" => [$dogStoryEbookId, $cookbookId],
21.    "creditCard" => [
22.        "number" => "4242424242424242",
23.        "validTillMonth" => 12,
24.        "validTillYear" => 2028,
25.        "cvc" => 123
26.    ]
27. ]));
28.
29. echo sprintf("Orders history:\n%s\n", $orderController->getOrders());

```

Listing 8.

```

1. CREATE TABLE ebooks (
2.     ebook_id INT PRIMARY KEY,
3.     title VARCHAR(255),
4.     content TEXT,
5.     price FLOAT
6. );
7. CREATE TABLE orders (
8.     order_id VARCHAR(36) PRIMARY KEY,
9.     email VARCHAR(255),
10.    credit_card JSON,
11.    related_ebook_ids JSON,
12.    price FLOAT,
13.    occurred_at TIMESTAMP
14. );
15. CREATE TABLE promotions (
16.     email VARCHAR(255) PRIMARY KEY,
17.     amount_of_orders INT
18. );

```

We have everything set up. The demo project (Listing 8) contains migrations to set up the database structure.

We are using the code in Listing 8 to replace code usually called via HTTP—we are just calling from CLI. You may treat this code as a test case. We will be working on internals, and this code will stay untouched, which means the so-called clients of our API will continue to work as before, even after our series of refactoring.

💡 Applications built with DDD methodology are agnostic to the environment they are executed in (HTTP, CLI, or RabbitMQ), as they focus on solving business problems, not the execution platform.

Now we can call this code shown in Listing 9 from CLI.

After execution, we get the following results:

Pushing Toward Domain Model

```

Making order for two books
Payment was performed for 35 using
credit card number: 4242424242424242,
valid till: 12/2028 and cvc: 123
Email sent to johnybravo@o3.en
with given titles:
Happy Dog Story,Cookbook - Home Recipes
Orders history:
[[
    "order_id":"51593442-a22a-481a-8454-c112f3fca3e7",
    "email":"johnybravo@o3.en",
    "credit_card_number":"4242424242424242",
    "related_ebook_ids":["1,2]",
    "price":"35",
    "occurred_at":"2022-02-20 11:01:00"
]]

```


This application was so successful that we have been running the business for 3 years. We've added multiple new functionalities. Some developers left, and a new one joined. Business demanded speed in development; however, changing the code became problematic. It takes a lot of time to analyze what actually happens; changing the code often crashes unrelated modules, and we are often afraid to change something. The codebase became part of daily jokes.

Let's go back to the past and help our future selves and other developers. Let's rethink how we can build this application differently. How can we make it readable, solid, and maintainable?

Do you remember invariants like price must be higher than 0 or credit card must be valid? We put if statements in our services to verify our invariants. Let's consider these questions:

- If we validated email in one service, are we sure it's still valid when we use it and pass it to other services? How do we handle situations where we use email in different places in our codebase?
- How can we be sure about the specific format of received data, like credit card numbers containing spaces or not?
- How do we know what structure to deal with when passing an array? Are we sure that all data in the array is correct?

We can't be sure about the above points until we analyze or debug them. Because of that, some strange things can happen in the code, which in later stages become invariants themselves, like the `EmailService` starting to handle an incorrect email with if statements because we are no longer sure that

it is receiving a valid email. Or, we begin formatting a credit card number inside the `PaymentGateway` to handle the situation when it contains spaces.

💡 *Fixing an incorrect state is not a business invariant—it's a technical problem we solve in code. These fixes can really blur what's important in the code, which can blur the Problem Space.*

We may encapsulate invariants like `Email`, `CreditCard`, `Price` in classes. We would validate correctness during the construction; we would throw an exception if it's invalid. Doing so is a pattern from Solution Space called `Value Objects`.

💡 *Value Objects contain a given state and a list of invariants that protect us since it's always valid. They also may contain related business logic.*

If we pass the string to a method, we can be sure that this variable will still hold the same value when this method finishes execution. This is not the same with classes. If we pass the variable containing `Email` to `→setEmail()`, we may have a different email inside after method execution. A mutable state may be really tricky to debug. That's why `Value Objects` are immutable.

💡 *Value Objects are immutable; the inner state is always the same as it was constructed. If you want to change the state, you need to create a new instance of this Value Object.*



You're the Team Lead—Now What?

Whether you're a seasoned lead developer or have just been "promoted" to the role, this collection can help you nurture an expert programming team within your organization.

After reading this book, you'll understand what processes work for managing the tasks needed to turn a new feature or bug into deployable code.

Order Your Copy
<http://phpa.me/devlead-book>

Listing 10.

```

1. final class Price
2. {
3.     public function __construct(
4.         public readonly float $amount
5.     ) {
6.         if ($amount < 0) {
7.             throw new \InvalidArgumentException(
8.                 "Price must be higher or equal tto 0");
9.         }
10.    }
11.
12.    public static function zero(): self
13.    {
14.        return new self(0);
15.    }
16.
17.    public function add(self $price): self
18.    {
19.        return new self($this->amount + $price->amount);
20.    }
21.
22.    public function multiply(float $multiplyBy): self
23.    {
24.        return new self($this->amount * $multiplyBy);
25.    }
26. }

```

Listing 11.

```

1. class Ebook
2. {
3.     private int $ebookId;
4.     private string $title;
5.     private string $content;
6.     private Price $price;
7.
8.     public function __construct(array $data)
9.     {
10.        if (strlen($data["title"]) <= 0) {
11.            throw new \InvalidArgumentException("...");
12.        }
13.        if (strlen($data["content"]) < 10) {
14.            throw new \InvalidArgumentException("...");
15.        }
16.
17.        $this->ebookId = $data["ebookId"];
18.        $this->title = $data["title"];
19.        $this->content = $data["content"];
20.        $this->price = $data["price"];
21.    }
22.
23.    public function getPrice(): Price
24.    {
25.        return $this->price;
26.    }
27.
28.    public function getTitle(): string
29.    {
30.        return $this->title;
31.    }
32. }

```

We are using clear SQL to insert data directly into the database. Doing so does not protect our invariants, however. If we have SQL all over the codebase, we may find ourselves in a place where somebody will accidentally store incorrect data, like an invalid email address. If we want to be sure about the validity of the data, we would need to apply the logic of our invariants on the database level. That is not always possible and may require writing constraints and database triggers to protect it.

What DDD proposes instead are Aggregates. An aggregate is a higher-level abstraction than the database layer, built into code. An aggregate only allows changes that keep the state valid; it may use Value Objects. It exposes a set of methods (behaviors) that protect the invariants inside.

💡 *Aggregates become our API for changes. In the case of modification, we will be working directly with them, not with the storage behind them.*

If you're familiar with Doctrine, you may consider an aggregate as entity-rich in behavior. If you're familiar with Eloquent you may consider it model-rich in behavior. However, take into consideration, that a Model in DDD has a wider meaning.

All of these patterns are part of the Domain Model. Domain is a term for a Problem Space that we are solving. In our case, we are in a Shopping/E-Commerce Domain. Model is within the Solution Space, where actual implementation that solves this Problem Space exists. Aggregates, Value Object, and Repositories are patterns from the Solution Space, and now we will be applying them to our codebase.

2nd Version—Model

Let's start by defining the model (Listing 11) for Ebook.

We have defined that price will always be greater than 0. Since we will be passing Price around the codebase, it's worth encapsulating this logic inside a Value Object as shown in Listing 10.

Now let's create an Aggregate called Ebook. We will cover title and content length invariants inside Ebook.

We also need a way to store Ebook. Under the hood, we will be using the same database table. However, we will encapsulate the logic for storing Ebook in the Repository. See Listing 12.

💡 *The repository pattern from solution space acts as a bridge covering all the details related to mapping your aggregate to storage. It exposes methods that allow for saving and fetching the aggregate.*

Listing 12.

```

1. // 1
2. use Ecotone\Messaging\Gateway\Converter\Serializer;
3.
4. class EbookRepository
5. {
6.     public function __construct(
7.         private Connection $connection,
8.         private Serializer $serializer
9.     ) { }
10.
11.     // 2
12.     public function save(Ebook $ebook): void
13.     {
14.         $data = $this->serializer->convertFromPHP(
15.             $ebook,
16.             MediaType::APPLICATION_X_PHP_ARRAY);
17.
18.         $this->connection->insert("ebooks",
19.             $this->convertCamelCaseToUnderscores($data));
20.     }
21.
22.     // 3
23.     public function getById(int $ebookId): Ebook
24.     {
25.         $data = $this->connection->executeQuery(sprintf(<<<SQL
26.             SELECT * FROM %s WHERE %s = :id
27.             SQL, "ebooks", "ebook_id"),
28.             ["id" => $ebookId]
29.         )->fetchAssociative();
30.
31.         if (!$data) {
32.             throw new \InvalidArgumentException("Ebook not found");
33.         }
34.
35.         return $this->serializer->convertToPHP(
36.             $this->underscoresToCamelCase($data),
37.             MediaType::APPLICATION_X_PHP_ARRAY,
38.             Ebook::class
39.         );
40.     }
41.
42.     ...
43. }

```

Listing 13.

```

1. class PriceConverter
2. {
3.     #[Converter]
4.     public function convertFrom(Price $price): int
5.     {
6.         return $price->amount;
7.     }
8.
9.     #[Converter]
10.    public function convertTo(int $price): Price
11.    {
12.        return new Price($price);
13.    }
14. }

```

1. We are using Ecotone's Serializer, which integrates with the solid and stable JMS Serializer. It allows us to serialize/deserialize classes easily.

💡 *Repository powered by Serializer does the database mapping. You can replace this implementation with your own Serializer or use an ORM like Doctrine or Eloquent.*

1. save takes the Ebook Aggregate and stores it inside the database

2. getById uses the ebookId to fetch the Ebook Aggregate from the database

As we will be converting int directly to Price, we need to tell our Ecotone's Serializer how to do it. We do this in Listing 13 by implementing a Converter.

I have dropped EbookService, as it does not do much now, and as shown in Listing 14 we can move the remaining logic inside EbookController.

Now we can build the Order and Promotion Models.

Let's start with our Value Objects:

```
final class Email { // See implementation in github }
```

```
final class CreditCard { // See implementation in github }
```

And in Listing 15, our Order Aggregate.

And our Promotion Aggregate in Listing 16.

We need to register Converters for Email, DateTimeImmutable, and UuidInterface.

Repositories

Since relatedEbookIds and creditCard are arrays, we need to encode them before storing them in the database. See Listing 17.

Note that PromotionRepository (Listing 18) is different and returns a default in case it was not found by getById. Doing so will simplify our higher-level code.

Let's check our OrderService after introducing the model. See Listing 19.

3rd Version—Messaging

In general, you can work your way out in building models without a Framework. Having your model clean of external tooling supports maintainability. However, it may be worth it to use a Framework to some extent, as there is always code that can be reused and that needs to "be glued."

Listing 14.

```

1. class EbookController
2. {
3.     public function __construct(private EbookRepository $ebookRepository, private Serializer $serializer)
4.     {
5.     }
6.
7.     public function registerEbook(string $requestAsJson): void
8.     {
9.         $data = json_decode($requestAsJson, true, flags: JSON_THROW_ON_ERROR);
10.
11.         $data["price"] = new Price($data["price"]);
12.
13.         $this->ebookRepository->save(new Ebook($data));
14.     }
15.
16.     public function getEbook(string $ebookId): string
17.     {
18.         return $this->serializer->convertFromPHP(
19.             $this->ebookRepository->getById($ebookId),
20.             "application/json"
21.         );
22.     }
23. }

```

Listing 15.

```

1. class Order
2. {
3.     private UuidInterface $orderId;
4.     private Email $email;
5.     private CreditCard $creditCard;
6.     /** @var int[] */
7.     private array $relatedEbookIds;
8.     private Price $price;
9.     private \DateTimeImmutable $occurredAt;
10.
11.     /** full implementation in github repository */
12. }

```

Ecotone understands that, and that's why it goes a different way. The main tenant is that business code should be independent of Framework as much as possible. You will rarely need to implement or extend classes from Ecotone. In most cases, you will annotate code with Attributes, and Ecotone will glue things together on your behalf. Doing so means the code stays clean of the framework and boilerplate, creating a clean and solid solution.

Let's take a look at `placeOrder()` from the `OrderService` class in Listing 20.

If we take a close look, we will see that this process performs actions on 3 Services with one call.

- Transaction to our local database to store Order and Promotion
- Sending an Email using an external Provider
- Charging client's credit card using Payment Gateway

Let's consider three scenarios:

Listing 16.

```

1. class EbookController
2. {
3.     public function __construct(
4.         private EbookService $ebookService
5.     ) {}
6.
7.     public function registerEbook(string $requestAsJson): void
8.     {
9.         $data = json_decode($requestAsJson, true,
10.             flags: JSON_THROW_ON_ERROR);
11.
12.         $this->ebookService->registerEbook($data);
13.     }
14.
15.     public function updateEbook(string $requestAsJson): void
16.     {
17.         $data = json_decode($requestAsJson, true,
18.             flags: JSON_THROW_ON_ERROR);
19.
20.         $this->ebookService->updateEbook($data);
21.     }
22.
23.     public function getEbook(string $ebookId): string
24.     {
25.         return json_encode(
26.             $this->ebookService->getEbookById($ebookId));
27.     }
28. }

```


Listing 17.

```

1. class OrderRepository
2. {
3.     public function __construct(private Connection $connection, private Serializer $serializer)
4.     {
5.     }
6.
7.     public function save(Order $order): void
8.     {
9.         $data = $this->serializer->convertFromPHP($order, MediaType::APPLICATION_X_PHP_ARRAY);
10.        $data["relatedEbookIds"] = \json_encode($data["relatedEbookIds"]);
11.        $data["creditCard"] = \json_encode($data["creditCard"]);
12.
13.        $this->connection->insert("orders", $this->convertCamelCaseToUnderscores($data));
14.    }
15.
16.    ( /** To see full implementation check github repository */ )
17. }

```

Listing 18.

```

1. class PromotionRepository
2. {
3.     public function getById(Email $email): Promotion
4.     {
5.         $data = $this->connection->executeQuery(
6.             sprintf(<<<SQL
7.                 SELECT * FROM %s WHERE %s = :email
8.             SQL, "promotions", "email"),
9.             ["email" => $email->address]
10.        )->fetchAssociative();
11.
12.        if (!$data) {
13.            return new Promotion($email);
14.        }
15.
16.        return $this->serializer->convertToPHP(
17.            $this->underscoresToCamelCase($data),
18.            MediaType::APPLICATION_X_PHP_ARRAY,
19.            Promotion::class
20.        );
21.    }
22.
23.    ( /** full implementation in github */ )
24. }

```

- What if Payment Gateway is down?

In our flow, if the PaymentGateway were down, we would lose the Order. A better solution would be to retry and keep the order.

- What if Email Provider is down? The changes we made in the database would roll back; however, PaymentGateway has already charged the client. That could make the client angry, receiving nothing after payment.

- What if our database would not respond on commit? Then we would end up charging the client and sending an email. However, we would lose the track of the Order.

Listing 19.

```

1. class OrderService
2. {
3.     public function placeOrder(array $data): void
4.     {
5.         $promoRepository = $this->promotionRepository;
6.         $paymentGateway = $this->paymentGateway;
7.         $emailService = $this->emailService;
8.         $relatedEbooks = [];
9.         foreach ($data["ebookIds"] as $ebookId) {
10.             $relatedEbooks[] =
11.                 $this->ebookRepository->getById($ebookId);
12.         }
13.
14.         $price = Price::zero();
15.         foreach ($relatedEbooks as $ebook) {
16.             $price = $price->add($ebook->getPrice());
17.         }
18.
19.         $promotion = $promoRepository->getById($data["email"]);
20.         $data["price"] = $promotion->isGrantedToPromotion() ?
21.             ($price->multiply(0.9)) : $price;
22.
23.         $this->connection->beginTransaction();
24.         try {
25.             $this->orderRepository->save(new Order($data));
26.             $promotion->increaseOrderAmount();
27.             $promoRepository->save($promotion);
28.             $paymentGateway->performPayment($data["creditCard"],
29.                 $price);
30.             $emailService->sendTo($data["email"], $relatedEbooks);
31.             $this->connection->commit();
32.         } catch (\Throwable $exception) {
33.             $this->connection->rollBack();
34.
35.             throw $exception;
36.         }
37.     }
38. }

```

Listing 20.

```

1. public function placeOrder(array $data): void
2. {
3.     $relatedEbooks = [];
4.     foreach ($data["ebookIds"] as $ebookId) {
5.         $relatedEbooks[] = $this->ebookRepository->getId($ebookId);
6.     }
7.
8.     $price = Price::zero();
9.     foreach ($relatedEbooks as $ebook) {
10.        $price = $price->add($ebook->getPrice());
11.    }
12.
13.    $promotion = $this->promotionRepository->getId($data['email']);
14.    $data["price"] = $promotion->isGrantedToPromotion() ? ($price->multiply(0.9)) : $price;
15.
16.    $this->connection->beginTransaction();
17.    try {
18.        $this->orderRepository->save(new \Ecotone\App\Model\Order\Order($data));
19.        $promotion->increaseOrderAmount();
20.        $this->promotionRepository->save($promotion);
21.        $this->paymentGateway->performPayment($data["creditCard"], $price);
22.        $this->emailService->sendTo($data["email"], $relatedEbooks);
23.        $this->connection->commit();
24.    } catch (\Throwable $exception) {
25.        $this->connection->rollBack();
26.        throw $exception;
27.    }
28. }

```

Those things happen, and unfortunately, they mostly occur when the software is in production. So how can we deal with this?

Messaging to the Rescue

Messaging is like the Post Office, which aims to deliver a letter (Message) to a given person. When we send a letter, we take it to the Post Office, and they take care of delivering it. In the meantime, we can focus on different things.

Messaging works the same way, and this is what provides great stability. So let's check how it can help in our example by making OrderService (Listing 21) more stable.

Right now, we are publishing the event OrderWasPlaced after we save the Order. And we have provided an Event Handler that will be called whenever we publish the event.

After Payment is complete, we publish another event, OrderPaymentWasSuccessful, which executes sendTo to send an email and increasePromotion to increase the current promotion count.

This code provides a nice separation of concerns, as each EventHandler does its own thing. However, if you execute this code, it will still run in a synchronous way. We have a situation where the postman has to deliver the exact same letter with instructions to two or more recipients. Now imagine the postman has to deliver the exact same letter with

instructions to three recipients. The first recipient reads it, performs the instructions, and gives it back. At the second house, the recipient reads it, fails to handle the instructions, and keeps the letter.

We are not in a good situation. Instead of depending on each of the recipients, we will deliver a copy of the letter to each of them. We will also place the message inside an inbox—even if they are not available, they will still get it. If they have problems performing the instructions in the letter, they put it back in the inbox and handle it later, which will not affect any other recipient.

So let's examine how we can extend the code to make it possible. We have the EventBus (Postman), delivering a copy of the letter (Message) to the inbox (Message Channel) where the recipient (Event Handler) can take it out and perform actions.

We will use our database as the inbox for storing our letters (Messages). First, we need to register a connection to make use of Ecotone and then we will register our inbox (Message Channel):

```

DbalConnectionFactory::class => DbalConnectionFactory::create($connection)
or new Connection("pgsql://ecotone:secret@localhost:5432/ecotone")

```

```

class MessagingConfiguration {
    #[ServiceContext]
    public function registerMessageChannel() {
        return DbalBackedMessageChannelBuilder::create("order_channel");
    }
}

```

Listing 21.

```

1. class OrderService
2. {
3.     public function placeOrder(array $data): void
4.     {
5.         $relatedEbooks = [];
6.         foreach ($data["ebookIds"] as $ebookId) {
7.             $relatedEbooks[] = $this->ebookRepository->getById($ebookId);
8.         }
9.
10.        $price = Price::zero();
11.        foreach ($relatedEbooks as $ebook) {
12.            $price = $price->add($ebook->getPrice());
13.        }
14.
15.        $promotion = $this->promotionRepository->getById($data['email']);
16.        $data["price"] = $promotion->isGrantedToPromotion() ? ($price->multiply(0.9)) : $price;
17.
18.        $order = new \Ecotone\App\Model\Order\Order($data);
19.        $this->orderRepository->save($order);
20.
21.        // 1
22.        $this->eventBus->publish(new OrderWasPlaced($order->getOrderId()));
23.    }
24.
25.    // 2
26.    #[EventHandler]
27.    public function performPayment(OrderWasPlaced $event, PaymentGateway $paymentGateway): void
28.    {
29.        $order = $this->orderRepository->getById($event->orderId);
30.        $creditCard = $order->getCreditCard();
31.
32.        $paymentGateway->performPayment($creditCard, $order->getPrice());
33.
34.        $this->eventBus->publish(new OrderPaymentWasSuccessful($event->orderId));
35.    }
36.
37.    // 3
38.    #[EventHandler]
39.    public function sendTo(OrderPaymentWasSuccessful $event, EmailService $emailService): void
40.    {
41.        $order = $this->orderRepository->getById($event->orderId);
42.        $ebooks = array_map(fn(int $ebookId) => $this->ebookRepository->getById($ebookId), $order->getRelatedEbookIds());
43.
44.        $emailService->sendTo($order->getEmail(), $ebooks);
45.    }
46.
47.    // 3
48.    #[EventHandler]
49.    public function increasePromotion(OrderPaymentWasSuccessful $event): void
50.    {
51.        $order = $this->orderRepository->getById($event->orderId);
52.
53.        $promotion = $this->promotionRepository->getById($order->getEmail());
54.        $promotion->increaseOrderAmount();
55.        $this->promotionRepository->save($promotion);
56.    }
57. }

```

```
#[EventHandler]
public function performPayment(OrderWasPlaced $event, PaymentGateway $paymentGateway): void
{
    // (...)
    $paymentGateway->performPayment($creditCard, $order->getPrice());
    $this->eventBus->publish(new OrderPaymentWasSuccessful($event->orderId));
}
```

Now we can make use of it as shown in Listing 22.

Besides providing the Asynchronous attribute with the Message Channel name, we also provide the endpointId. The endpointId is the address of the recipient's house.

Add the following command at the end of example.php to execute Postman: \$application->run("order_channel");

If you are using Symfony or Laravel, use the CLI Command ecotone:run_order_channel.

💡 Ecotone provides retries in case a message fails during processing, and enables the possibility of storing the message in the database in case it fails a given amount of times. If you want to know more, check the documentation².

We have a few more things to clean up before our code can focus only on the problem space we are working on.

We have Converters for all our Value Objects, yet we use them only while fetching from the database. In the Controllers, we are building them manually. We could also benefit from dropping arrays and using classes to get more visibility about the data we are dealing with.

Listing 23 shows what we do when we are placing an order.

Let's create a class (Listing 24) that will wrap \$data parameters.

And let's mark our OrderService with the CommandHandler attribute and provide PlaceOrder as the first parameter.

```
#[CommandHandler("placeOrder")]
public function placeOrder(PlaceOrder $command): void
{
    (...)
}
```

Command Handler is a way to enable messaging for a given method. We can send a letter (Command) and let Postman do extra steps before it gets to the Recipient (Command Handler), like converting types.

Now we can execute it from the Controller. See Listing 25.

No conversions in the code mean much cleaner code. Let's look at one more thing that exposes too much information and could be done much more easily.

```
public function getOrders(): string
{
    return $this->serializer->convertFromPHP(
        $this->orderRepository->getAll(),
        "application/json"
    );
}
```

In our OrderRepository, let's add a getAll() Query Handler.

```
/**
 * @return Order[]
 */
#[QueryHandler("order.getAll")]
public function getAll(): array
```

Doing so means we expose the method for querying using order.getAll.

2 documentation: <https://docs.ecotone.tech>

Listing 22.

```
1. #[Asynchronous("order_channel")]
2. #[EventHandler(endpointId: "performPayment")]
3. public function performPayment(OrderWasPlaced $event, PaymentGateway $paymentGateway): void
4.
5. {...}
6.
7. #[Asynchronous("order_channel")]
8. #[EventHandler(endpointId: "sendTo")]
9. public function sendTo(OrderPaymentWasSuccessful $event, EmailService $emailService): void
10.
11. {...}
12.
13. #[Asynchronous("order_channel")]
14. #[EventHandler(endpointId: "increasePromotion")]
15. public function increasePromotion(OrderPaymentWasSuccessful $event): void
```


Listing 23.

```

1. public function placeOrder(string $requestAsJson): void
2. {
3.     $data = json_decode($requestAsJson, true,
4.         flags: JSON_THROW_ON_ERROR);
5.     $data['email'] = new Email($data['email']);
6.     $data['creditCard'] = new CreditCard(
7.         $data['creditCard']['number'],
8.         $data['creditCard']['cvc'],
9.         $data['creditCard']['validTillYear'],
10.        $data['creditCard']['validTillMonth']
11.    );
12.
13.    $this->orderService->placeOrder($data);
14. }

```

Listing 24.

```

1. class PlaceOrder
2. {
3.     public readonly Email $email;
4.     public readonly CreditCard $creditCard;
5.     /**
6.      * @var int[]
7.      */
8.     public readonly array $bookIds;
9. }

```

Listing 25.

```

1. class OrderController
2. {
3.     public function __construct(
4.         private CommandBus $commandBus,
5.         private OrderRepository $orderRepository,
6.         private Serializer $serializer
7.     ) { }
8.
9.     public function placeOrder(string $requestAsJson): void
10.    {
11.        $this->commandBus->sendWithRouting("placeOrder", $requestAsJson, "application/json");
12.    }
13.
14.    (...)
15. }

```

```

public function getOrders(): string
{
    return $this->queryBus->sendWithRouting(
        "getAllOrders",
        expectedReturnedMediaType: "application/json"
    );
}

```

We call the `getAllOrders()` Query Handler and tell Ecotone that we expect it to return `application/json`.

Summary

We started with a clear SQL implementation. Then we introduced a domain model for more readability and maintainability. And in the last section, we brought in messaging to make the code even more maintainable and solid.

We can solve any problem with multiple solutions. However, it's worth bringing solutions that help us achieve long-term goals.

We want to have a solid foundation from good quality blocks, so we can feel safe and secure when we make changes on the upper floors.

If you want to check the full implementation of the project we were working on, visit the GitHub repository³.



I am Software Developer and Architect and a big enthusiast of DDD CQRS and Distributed Architectures. That enthusiasm finally pushed me into creating PHP framework - Ecotone, based on Messaging Architecture and principles of DDD, CQRS, and Event Sourcing. I love expanding and sharing my knowledge using blog articles and writing open-source software.
[@DariuszGafka](https://github.com/dgafka/php-architect-ebook-shop-demo)

³ GitHub repository:

<https://github.com/dgafka/php-architect-ebook-shop-demo>



The Web Developer's

SECRET WEAPON!

Exception, uptime, and cron monitoring, all in one place and easily installed in your web app. Deploy with confidence and be your team's devops hero.

Are exceptions *all* that keep you up at night?

Honeybadger gives you full confidence in the health of your production systems.

DevOps monitoring, for developers. *gasp!*

Deploying web applications at scale is easier than it has ever been, but monitoring them is hard, and it's easy to lose sight of your users.

Honeybadger simplifies your production stack by combining three of the most common types of monitoring into a single, easy to use platform.

Exception Monitoring

Delight your users by proactively monitoring for and fixing errors.

Uptime Monitoring

Know when your external services go down or have other problems.

Check-In Monitoring

Know when your background jobs and services go missing or silently fail.



Start Your Free Trial Today
<https://www.honeybadger.io/>



Event-Driven Programming

Chris Tankersley

One of the interesting things about programming is that when faced with a similar problem, developers tend to build the same solution. They may call their implementations different names, or there may be slight differences at a superficial level, but ultimately the architectural design is the same. At a base level, we tend to call these Patterns.

You may have heard of the term “Design Pattern.” This term is used to describe many different ways to structure object-oriented code. These design decisions were most famously cataloged in “Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The idea behind design patterns is that developers can notice a pattern and understand much of the “why” behind using the pattern. When you work with a database object, you should be able to tell if it is using a Data Mapper pattern or not and get some expectation of how it should work overall.

Humans are great at recognizing patterns, and the more code you come into contact with, the more you will notice these patterns. While we will not be diving deep into the wide range of patterns in PHP, the more you program and interact with other people, you will notice these patterns emerge even if you had no knowledge of a particular pattern when you programmed. This lays a foundation for common language developers can use.

One step above a design pattern is Programming Paradigms. Discussions about paradigms tend to exist at the language level. You get into terms like Event-Driven Programming, Concurrent Programming, and Object-Oriented. A language can contain many different paradigms, but the available paradigms help influence how the next level above this is handled.

The highest level of these patterns is an Architectural Pattern. These tend to exist in the ephemeral “problem needs solved” space. Things like using an Extract, Translate, and Load (ETL) architecture explains how to solve the problem of taking data from one place to another, but how exactly you do that is constrained by the paradigms of your language. Those paradigms influence possible design patterns you may see in code.

In PHP, we commonly see the architectural pattern called Action-Domain-Responder, or ADR. This pattern is similar to another called Model-View-Controller, or MVC. Due to the paradigms in PHP, the MVC that we typically see has been modified to work with PHP’s non-concurrent procedural object-oriented design decisions and has given rise to another architectural pattern called Web MVC. Inside the “Model” portion of this architecture pattern, we see design patterns like Active Record or Data Mappers.

Other languages may also use MVC or even Web MVC. In an asynchronous language like NodeJS, the Models may

follow a similar pattern to a Data Mapper but rely on one that utilizes callbacks to work with data instead of just directly returning the data. The key with all of this is that NodeJS, Ruby, and PHP developers can all use common terms and patterns to solve the same problem of building a dynamic website and still understand each other at a conceptual level. The actual implementations will look similar but not necessarily exact.

One paradigm that many developers tend to overlook in PHP is the Event-Driven Programming paradigm. This paradigm is not to be confused with Event Sourcing, which is a design pattern that deals with how to store transactional or historical data. Event-Driven Programming is a paradigm that discusses a system watching for things to happen and then deciding how to react to those events. As with many things, this paradigm gets shifted slightly due to other things in PHP, but it is there.

If you have worked with JavaScript, think of it as attaching an event listener to a button and waiting for a click event.

```
let el = document.getElementById('myButton');  
  
el.addEventListener('click', function(event) {  
    console.log('The button was clicked!');  
});
```

In the example, we grab an element with an id of “myButton.” We then tell the engine to listen for the “click” event, and when that happens, fire off the function. The JavaScript engine will constantly be looping in the background, waiting for events. As soon as the engine sees the button click, it tells anyone listening for that event to execute. Our function fires, and we get a nice little console message.

Many languages that deal with graphical user interfaces have an event-driven paradigm. The idea is the same as with JavaScript. The main process constantly loops, **waiting for events, and when an event happens, other associated code then fires. The associated code never executes if the event never happens. This paradigm is very different than Procedural programming, a paradigm where each line of code is executed one after the other. PHP, incidentally, has a procedural paradigm.

How can we take advantage of an event-driven paradigm in a procedural paradigm-based language? By cheating, just a little bit. And some design patterns.



Observer Pattern

The core idea of Event-Driven Design is that parts of your code watch for things to happen. We can look at different design patterns to see how to solve this problem. One of the more common design patterns that shows this off is the Observer Pattern.

The Observer Pattern is where several objects, called Observers, watch another object, called a Subject, and wait for the Subject to notify them that its state has changed. This pattern is so common that it is baked into PHP as the `\SplSubject` and `\SplObserver` interfaces. We can write classes that implement these interfaces to set up an event-driven system.

Let us imagine that we are building out a new content management system, and as part of that, we need to register new users. We want developers to be able to write custom

actions that happen when the user is registered. One way to do this is to have a service that creates users and allows other objects to watch for that event.

Since this is an event-driven system, let's create a subject that we want to watch (Listing 1).

The `\SplSubject` interface requires us to implement three methods—`attach()`, `detach()`, and `notify()`. The `attach()` method allows observers to let a subject know that they want to be notified when events happen. `detach()` allows observers to stop listening for events from the Subject, and `notify()` is meant to let any attached observers know that an event is happening.

We also add a `createUser()` service. This service takes the details of a new user and creates them in the system. It will then set the state of the service to “new,” store the user and notify all the observers. They can use this information to do whatever they want and decide if they want to react to the current state of our service.

Next, we need to have something that wants to execute when that event fires. We can create a class that implements `\SplObserver`. The observer class we create just needs a single method named `update()`. This method takes an `\SplSubject` as a parameter, so in our case, we will be expecting a `NewUserEvent` subject to be passed in.

Listing 1.

```
1. class UserService implements \SplSubject
2. {
3.     protected \SplObjectStorage $observers;
4.     public ?string $state;
5.     public User $user;
6.
7.     public function __construct()
8.     {
9.         $this->observers = new \SplObjectStorage();
10.    }
11.
12.    public function createUser(array $data): User
13.    {
14.        $user = new User($data);
15.        $user->save();
16.
17.        $this->state = 'new';
18.        $this->user = $user;
19.        $this->notify();
20.
21.        $this->state = null;
22.        return $user;
23.    }
24.
25.    public function attach(\SplObserver $observer): void
26.    {
27.        $this->observers->attach($observer);
28.    }
29.
30.    public function detach(\SplObserver $observer): void
31.    {
32.        $this->observers->detach($observer);
33.    }
34.
35.    public function notify(): void
36.    {
37.        foreach ($this->observers as $observer) {
38.            $observer->update($this);
39.        }
40.    }
41. }
```

```
class SendNewUserEmail implements \SplObserver
{
    public function update(\SplSubject $subject): void
    {
        $email = new Email('New user registration',
            $subject->user->getEmail());
        // ... Build the e-mail and send it
    }
}
```

Now we can create our service, attach our observer, and create a new user.

```
$service = new UserService();
$service->attach(new SendNewUserEmail());
$service->createUser([
    'username' => 'dragonmantank',
    'email' => 'myemail@domain.com'
]);
```

Since we attached the `SendNewUserEmail` object to the service, when `notify()` is called, the service is passed to the observer, and they can react to it. If someone also wants to send an SMS message, or add in an additional database log entry, they can simply attach new observers to the service class.

Event Dispatching

One of the downsides of the `\SplObserver` and `\SplSubject` classes is their design. A `\SplSubject` only has a single `notify()` method that takes no arguments, so all of the observers can only really watch for a single event. Something like a service layer or a Repository layer (which handles actual database



interactions) may want to broadcast multiple events without having to change states.

This use-case is where a design pattern known as Event Dispatching comes into play. Instead of individual subjects keeping track of their observers, a separate entity known as an Event Dispatcher keeps track of them. Other objects can then fire off an “event” or some action that happens through the Event Dispatcher. It is very similar to how the Observer pattern works but allows a single object to handle multiple events and centralizes where this logic is handled.

The Event Dispatcher pattern is so common it spawned PSR-14: Event Dispatcher¹. This PSR details how to create an interoperable event Dispatcher than can be used by multiple systems. It is a bit more complicated than the simple Observer pattern, but this complication allows much more flexibility. It provides:

- **Event**—An Event is a message produced by an *Emitter*. It may be any arbitrary PHP object.
- **Listener**—A Listener is any PHP callable that expects to be passed an Event. Zero or more Listeners may be passed to the same Event. A Listener MAY enqueue some other asynchronous behavior if it so chooses.
- **Emitter**—An Emitter is any arbitrary code that wishes to dispatch an Event. This is also known as the “calling code”. It is not represented by any particular data structure but refers to the use case.
- **Dispatcher**—A Dispatcher is a service object that is given an Event object by an Emitter. The Dispatcher is responsible for ensuring that the Event is passed to all relevant Listeners but MUST defer determining the responsible listeners to a Listener Provider.
- **Listener Provider**—A Listener Provider is responsible for determining what Listeners are relevant for a given Event but MUST NOT call the Listeners themselves. A Listener Provider may specify zero or more relevant Listeners.

(From the PSR-14 documentation)

At a very basic level, we need to create something to hold the listeners for the events in our system, something to dispatch events, and events. Our code can then call the dispatcher at any point to invoke an event. This dispatcher can be passed to many objects and can serve as a central place for us to encapsulate working with events.

Creating an object to keep track of listeners does not require much code. Let's create a simple class (Listing 2) that implements `Psr\EventDispatcher\ListenerProviderInterface`.

All this class does is take the name of the event class we want to listen for via `addListener()` and what code to invoke. Our Dispatcher, which we will detail in a moment, will then ask which of the listeners are listening for a particular event.

Listing 2.

```
1. class ListenerProvider implements ListenerProviderInterface
2. {
3.     protected array $listeners = [];
4.     public function addListener(string $eventType,
5.         callable $listener)
6.     {
7.         $this->listeners[$eventType][] = $listener;
8.     }
9.
10.    public function getListenersForEvent(
11.        object $event): iterable
12.    {
13.        return $this->listeners[$event::class];
14.    }
15. }
```

Speaking of our Dispatcher, the code for it is also fairly short as shown in Listing 3. It gets a ListenerProvider and will wait for some other object to dispatch an event through it.

When an event is dispatched, we ask our listener provider for all the listeners that are waiting for the incoming event, and then we pass that event to each listener. PSR-14 also has the concept of a “stoppable” event or an event that is forced to stop processing if something happens. For example, if we cannot save the user to the database, we should not process any other events that may depend on the database entry.

Now, all we need is an event and something to react to that event. For the sake of simplicity, we'll make a very simple event that just takes a user, and the listener will echo out the user's name. See Listing 4.

How would we use this with our UserService from before? We can just pass the Dispatcher into the service and remove

Listing 3.

```
1. class Dispatcher implements EventDispatcherInterface
2. {
3.     public function __construct(
4.         protected ListenerProviderInterface $listeners
5.     ) { }
6.
7.     public function dispatch(object $event)
8.     {
9.         $listeners = $this->listeners
10.            ->getListenersForEvent($event);
11.         foreach ($listeners as $listener) {
12.             if ($event instanceof StoppableEventInterface &&
13.                 $event->isPropagationStopped())
14.             {
15.                 return;
16.             }
17.             $listener($event);
18.         }
19.     }
20. }
```

¹ PSR-14: Event Dispatcher: <https://www.php-fig.org/psr/psr-14/>



Listing 4.

```
1. class CreateUserEvent
2. {
3.     public User $user;
4. }
5.
6. $listenerProvider = new ListenerProvider();
7. $listenerProvider->addListener(
8.     CreateUserEvent::class, function (object $event) {
9.         echo $event->user->username;
10.     });
11.
12. $dispatcher = new Dispatcher($listenerProvider);
13. $event = new CreateUserEvent();
14. $event->user = new User();
15. $event->user->username = 'dragonmantank';
16. $dispatcher->dispatch($event);
```

all the \SplSubject code. When we create a new user, we just dispatch a new event as shown in Listing 5.

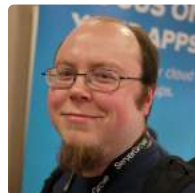
By moving all the event handling code to the Dispatcher, we clean up a lot of extra code and complexity from the UserService class. Now, we can also better support multiple events, like having events for when a user is updated or deleted. We just create new event classes with the relevant information and pass them to the Dispatcher.

Reacting Versus Extending

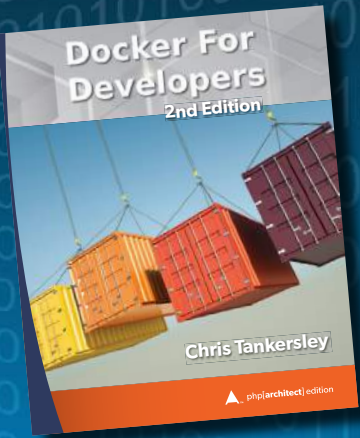
Hopefully, this gives you a good idea of a different way to allow a system to expand its capabilities. If you are writing code that will need to be extended by users, but you know the additions will not fit in the confines of inheritance or composition, an event-driven design might be for you. These types of systems can be extremely versatile. WordPress's hook system is a great example of an event-driven architecture that is hugely powerful.

Listing 5.

```
1. class UserService
2. {
3.     public function __construct(
4.         protected EventDispatcherInterface $dispatcher) {}
5.
6.     public function createUser(array $data): User
7.     {
8.         $user = new User($data);
9.         $user->save();
10.        $event = new CreateUserEvent();
11.        $event->user = $user;
12.        $this->dispatcher->dispatch($event);
13.    }
14. }
15.
16. $listenerProvider = new ListenerProvider();
17. $listenerProvider->addListener(CreateUserEvent::class,
18.     function (object $event) {
19.         echo $event->user->username;
20.     });
21.
22. $dispatcher = new Dispatcher($listenerProvider);
23. $service = new UserService($dispatcher);
24. $service->createUser($userData);
```



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](https://twitter.com/dragonmantank)



Docker For Developers is designed for developers looking at Docker as a replacement for **development environments** like virtualization, or devops people who want to see how to take an existing application and integrate Docker into their workflow.

This revised and expanded edition includes:

- Creating custom images
- Working with Docker Compose and Docker Machine
- Managing logs
- 12-factor applications

Order Your Copy
<http://phpa.me/docker-devs>

Random and Rare Failures

Edward Barnard

Designing for failure is difficult because it's usually not practical to predict every possible thing that could go wrong. Last month we got the preliminary "spadework" out of the way. This month we begin our first feature. This feature creates a mechanism for capturing those "random and rare" failures.

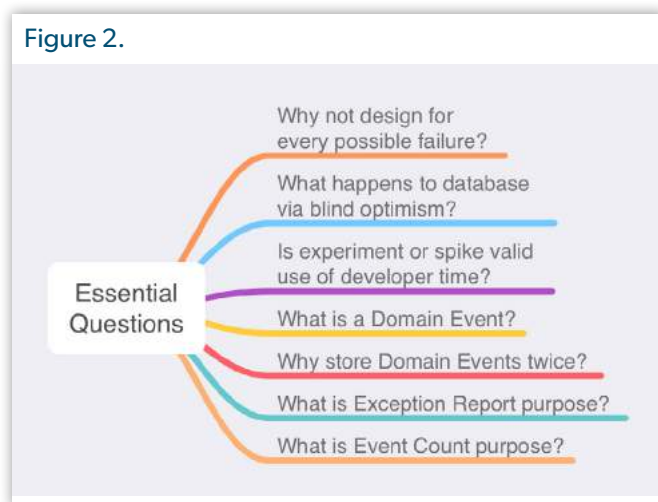
We introduced the concept behind this feature with "Designing for MySQL Transaction Failures" in December 2021 [php\[architect\]](#). All source code is available on GitHub at [ewbarnard/strategic-ddd](#)¹.

Essential Questions

Upon surviving this article, you should be able to answer these essential questions (Figure 2):

- Why can't we design for every possible failure?
- What can happen to a database in the path of blind optimism?
- Is an experiment or spike a valid use of developer time?
- What is a Domain Event in DDD?
- Why are we storing Domain Events twice?
- What is the Exception Report's purpose?
- What is the Event Count's purpose?

Figure 2.



¹ [ewbarnard/strategic-ddd](#):
<https://github.com/ewbarnard/strategic-ddd>



Random and Rare Failure

In theory, PHP projects generally have many points where something *could* fail, but that would rarely happen. For example, inserting a new row in a database table could fail for many reasons. But if your application has been running happily for weeks or months, we generally figure that particular section of code is unlikely to fail.

What if it *does* fail? If we created a separate contingency plan for every single line of code that could someday fail, we'd never get anything done. In practice, we often fall back to a general site error page and ask the user to try again. As the developer, you will likely not even know the problem ever happened.

Sometimes an error becomes a solid breakage and breaks 500 times per second around the clock. Until then, we often don't recognize the error, let alone come up with a contingency plan. Once the error becomes spectacular, we'll know what to do! But what about the rare failings that get missed?

Log files tend to be so noisy that they're unlikely to help with this sort of situation. Even with automated monitoring, it's challenging to hit the right level of notifying versus ignoring. With too many notifications, we are likely to start ignoring them.

Pollyanna Path

I've had numerous discussions in this area. They run something like this:

- If something never fails, we never need to worry about the failure.
- If it does fail, we have no idea why and therefore have no way to prevent it beforehand.



These discussions tend to come to the conclusions:

- If it happens, it happens, so why worry?
- Just code the “happy path,” assuming all is running correctly, and we’ll be fine.

I’ve also noticed that sometimes when the production database is large (many table rows, often with several years’ worth of data), foreign key constraints² are not possible.

Why is that? A foreign key constraint violation happens when a record in one table refers to the record in another table, but that other record doesn’t exist. It’s a form of data corruption. Even if it’s rare, for example, one problem per million table rows, MySQL won’t allow that table to enforce the foreign key constraint. It’s all or nothing.

At this point, in my view, we’ve been coding the “Pollyanna³ path” of blind optimism rather than the “happy path⁴” of primary functionality. The result is that the database is too expensive to fix and that we must therefore continue to live with the flawed situation. (There are ways to improve the situation, such as archiving data or carefully migrating to clean tables, but that’s outside the scope of this discussion.)

The question remains, what might we do about rare and random failures? Those “one in a million” failures can still be a problem. Is there anything we can do to help the situation?

There is. We’re going to shift focus for a moment and discuss an experiment. That experiment will bring us back to a solution.

History Trace or Event Trace

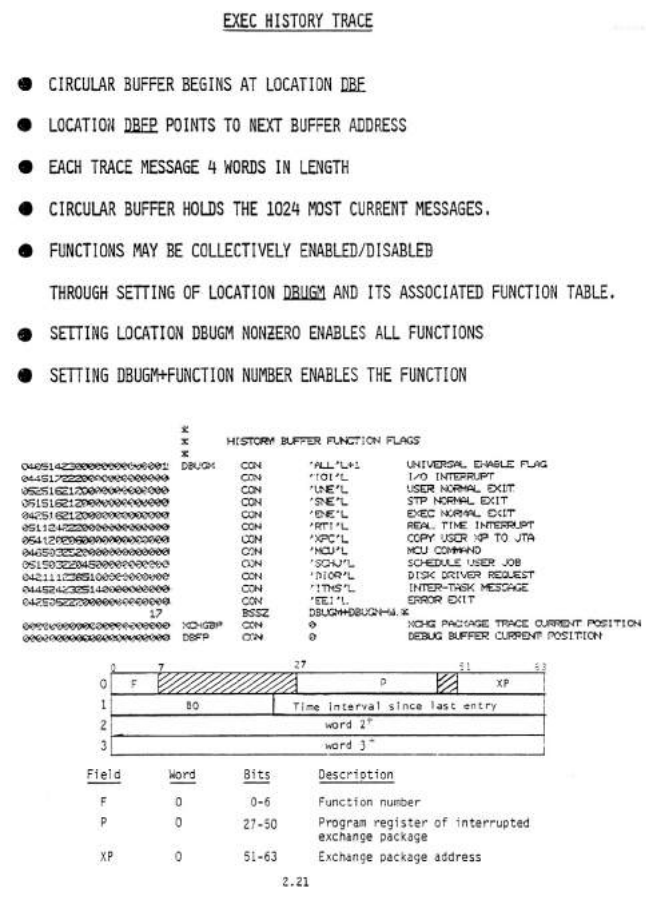
How important is a history trace (or event trace)? In my first full-time position, the target computer system only included 8 MBytes of main memory (RAM). It did not support virtual memory. Everything, including the user application, operating system, data buffers, etc., all had to fit within those 8 MBytes. Yet, the highly-experienced designers of that computer’s operating system dedicated a large chunk of operating system memory to an event trace (Figure 3). It retained the most recent 1024 events in memory. The objective was to answer questions like “what path did we follow to get to this point?”

Some of the best minds in our industry saw the importance of recording events. It was easier to figure out “what happened?” when the event trace exactly showed what happened when something went wrong.

That concept stayed with me. Might there be a way to apply the concept to “random and rare” failures we see in our web applications?

With a corrupt (or missing) database table row, “what happened?” can be a difficult question to answer. MySQL

Figure 3.



binary and relay logs⁵ might provide answers, but only if the database query shows the problem. We have no information to work with if the problem is in our PHP code.

Let’s write some experimental code and see where that takes us. Is that even allowed? What’s considered best practice concerning non-production code?

Experimentation

Scott Millett and Nick Tune, in *Patterns, Principles, and Practices of Domain-Driven Design* (2015) [Millett], p. 142, explain the importance of experimental designs:

A rich and useful model is a product of exploration and creativity. Experimentation is about looking at the code in a different way. If you find that coding is hard, you are probably doing something wrong. Don’t just stop at the first useful model you produce... You won’t get it right the first time. Experimentation and exploration fuel learning.

² foreign key constraints: <https://phpa.me/mysql-foreign-keys>

³ Pollyanna: <https://phpa.me/merriam-webster-pollyanna>

⁴ happy path: https://en.wikipedia.org/wiki/Happy_path

⁵ MySQL binary and relay logs:

<https://dev.mysql.com/doc/refman/8.0/en/replica-logs.html>



Both Scrum and Extreme Programming endorse the practice as a spike⁶:

A spike is a product development method originating with Extreme Programming that uses the simplest possible program to explore potential solutions. It is used to determine how much work will be required to solve or work around a software issue. Typically, a “spike test” involves gathering additional information or testing for easily reproduced edge cases.

A spike can be scheduled into a sprint like any other task. The result of this task will be knowledge to share with the team rather than production software to be released.

Domain Events

We’re about to create an experiment. Let’s first be clear about what our experiment is *not*. Domain-Driven Design (DDD) includes the Domain Events pattern. [Millet] explains (p. 405):

DDD practitioners have found that they can better understand the problem domain by learning about the events that occur within it—not just the entities. These events, known as domain events, will be uncovered during knowledge-crunching sessions with domain experts. Uncovering domain events is so valuable that DDD practitioners have innovated knowledge-crunching techniques to make them more event-focused using practices such as event storming.

Eric Evans, with *Domain-Driven Design Reference: Definitions and Pattern Summaries*, p. 17, summarizes Domain Events as:

Something happened that domain experts care about.

Our experiment, by contrast, will focus on events that we, as developers, care about. Martin Fowler, with *Domain Event*⁷, comes closer to our purpose:

Captures the memory of something interesting which affects the domain... The essence of a Domain Event is that you use it to capture things that can trigger a change to the state of the application you are developing... Capturing system stimuli...

If our objective is to answer the question “what went wrong?” we’ll need to record many Domain Events. We may need Fowler’s “stimuli” and Evans’ “something happened

that domain experts care about” definitions to get to the root cause or causes of the problem being investigated.

Overkill

Chapter 8 in Vaughn Vernon’s *Implementing Domain-Driven Design* explains that the event needs to be saved in a “local store” within the same transaction as the state change itself.

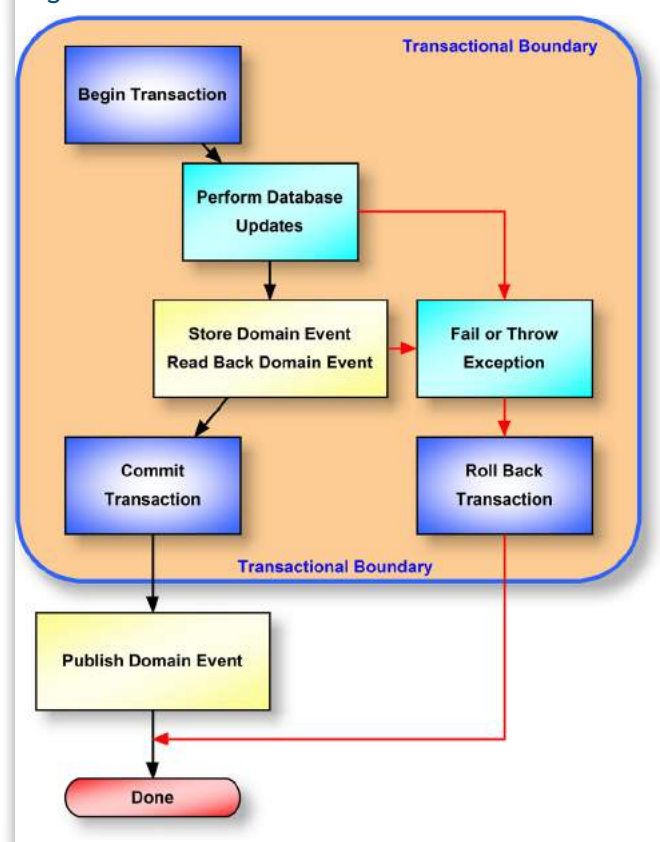
The event is a message broadcasting the fact that something interesting happened. However, Vernon explains, that message should only be published when the “something interesting” successfully happens. In other words, if our database update fails, the message should not be published.

In MySQL terms, we need this sequence (Figure 4):

1. Begin transaction.
2. Update the database (insert a record or whatever is needed).
3. Insert the Domain Event into a table in the same database (within the same database transaction—therefore using the same database connection).
4. If all was successful, commit the transaction. If not, roll back the transaction.

Finally, if, and only if, the transaction commits successfully, publish the domain event.

Figure 4.



⁶ spike: <https://w.wiki/4XP6>

⁷ Domain Event: <https://martinfowler.com/eaDev/DomainEvent.html>



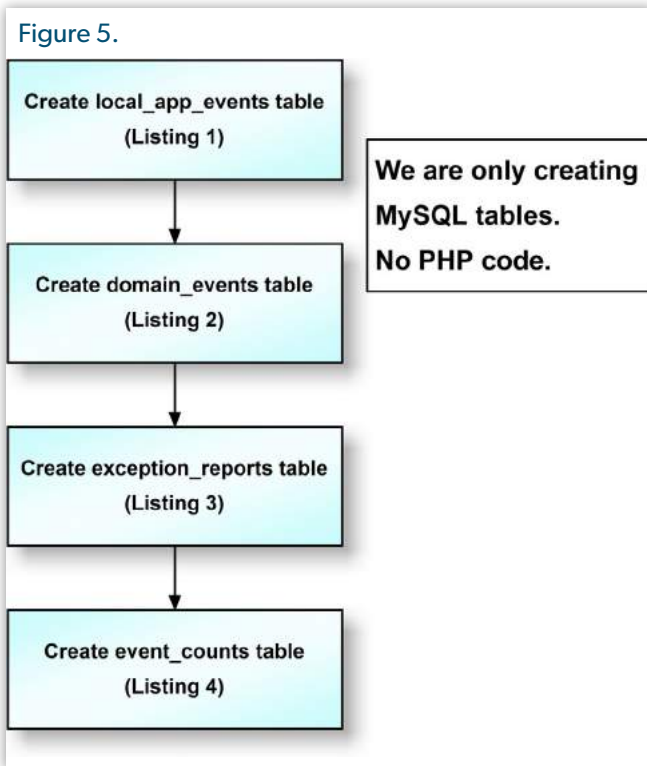
At first glance, this seems like overkill. We expect to be saving a very large number of domain events, and now we are saving them twice. We save a local copy as we're doing whatever we do to update the database, and then we publish the message, which probably includes saving it in some global event store.

Suddenly there's a lot more work involved. Why is this? PHP frameworks are usually smart enough to save related data inside the same transaction. But our domain event is *not* obviously related. That means we need to run the transaction manually.

In short, Vernon's advice means that *every* time I modify the database, I need to deal with a manual transaction, including commit or rollback.

The funny thing is, we just created an extremely useful mechanism. That mechanism will prove to be the key to capturing those "random and rare" failures. We start by building out support for Vernon's domain events.

Figure 5 shows the steps we'll be taking. All data description language⁸ (DDL) is available in the source code repository⁹.



Domain Event Tables

Listing 1 `local_app_events.ddl` shows the local events table structure.

The table includes both a primary key *and* a UUID because I expect to have several local events tables, one for each

⁸ data description language: <https://w.wiki/5BBy>

⁹ source code repository: <https://github.com/ewbarnard/strategic-ddd>

Listing 1.

```

1. CREATE TABLE `local_app_events`
2. (
3.     `id`            bigint unsigned NOT NULL AUTO_INCREMENT,
4.     `action`        varchar(255)    NOT NULL DEFAULT '',
5.     `subsystem`     varchar(255)    NOT NULL DEFAULT '',
6.     `description`   varchar(255)    NOT NULL DEFAULT '',
7.     `detail`        json             DEFAULT NULL,
8.     `event_uuid`    char(36)         NOT NULL,
9.     `when_occurred` timestamp(6)    NOT NULL,
10.    `created`        datetime         NOT NULL,
11.    `modified`       datetime         NOT NULL,
12.    PRIMARY KEY (`id`),
13.    UNIQUE KEY `event_uuid` (`event_uuid`)
14. ) ENGINE = InnoDB
15. DEFAULT CHARSET = utf8mb4
16. COLLATE = utf8mb4_0900_ai_ci;

```

database schema. Each local table would have a different table name to indicate which schema it resides in. I also find it easier to avoid confusion when I do *not* have two tables with the same name in two different schemas. It becomes more challenging to keep the PHP class names separated. Why create more difficulty when it's not necessary?

Once the event arrives in the global event store (Listing 2), the UUID could be used to track the event between tables. The UUID can also potentially be useful for tracking the domain event message through several subscribing processes.

The field `when_occurred` is a timestamp to the nearest microsecond that gets populated with the MySQL function `NOW(6)` during row insert. After inserting the row while still inside the transaction, we will read back the entire row and pass along the populated row (complete with generated primary key and microsecond timestamp) as the domain event.

Listing 2.

```

1. CREATE TABLE `domain_events`
2. (
3.     `id`            bigint unsigned NOT NULL AUTO_INCREMENT,
4.     `id_of_source`  bigint unsigned NOT NULL,
5.     `source_table`  varchar(255)    NOT NULL,
6.     `action`        varchar(255)    NOT NULL DEFAULT '',
7.     `subsystem`     varchar(255)    NOT NULL DEFAULT '',
8.     `description`   varchar(255)    NOT NULL DEFAULT '',
9.     `detail`        json             DEFAULT NULL,
10.    `event_uuid`    char(36)         NOT NULL,
11.    `when_occurred` timestamp(6)    NOT NULL,
12.    `created`        datetime         NOT NULL,
13.    `modified`       datetime         NOT NULL,
14.    PRIMARY KEY (`id`),
15.    UNIQUE KEY `event_uuid` (`event_uuid`),
16.    UNIQUE KEY `source_id` (`id_of_source`, `source_table`)
17. ) ENGINE = InnoDB
18. DEFAULT CHARSET = utf8mb4
19. COLLATE = utf8mb4_0900_ai_ci;

```



Exception Report

We'll store the failure's context in the Exception Report table when we detect one of those random and rare failures. See Listing 3.

Event Counts

We need some way to trigger failures while we're developing the exception report. Listing 4 shows the poorly-designed Event Counts table we'll use. We declared `when_counted` rows to be unique, but it is a `datetime` field accurate to the nearest second. If we store two rows in the same second, we should generate a MySQL failure based on violating the unique-key constraint.

Listing 3.

```
1. CREATE TABLE `exception_reports`
2. (
3.   `id`          int unsigned NOT NULL AUTO_INCREMENT,
4.   `description` varchar(255) NOT NULL DEFAULT '',
5.   `detail`      json          DEFAULT NULL,
6.   `created`     datetime      NOT NULL,
7.   `modified`    datetime      NOT NULL ON UPDATE CURRENT_TIMESTAMP,
8.   PRIMARY KEY (`id`)
9. ) ENGINE = InnoDB
10.  DEFAULT CHARSET = utf8mb4
11.  COLLATE = utf8mb4_0900_ai_ci;
```

Listing 4.

```
1. CREATE TABLE `event_counts`
2. (
3.   `id`          int unsigned NOT NULL AUTO_INCREMENT,
4.   `when_counted` datetime     NOT NULL COMMENT 'Intentional poor design',
5.   `event_count` bigint unsigned NOT NULL DEFAULT '0',
6.   `created`     datetime      NOT NULL,
7.   `modified`    datetime      NOT NULL,
8.   PRIMARY KEY (`id`),
9.   UNIQUE KEY `when` (`when_counted`)
10. ) ENGINE = InnoDB
11.  DEFAULT CHARSET = utf8mb4
12.  COLLATE = utf8mb4_0900_ai_ci;
```

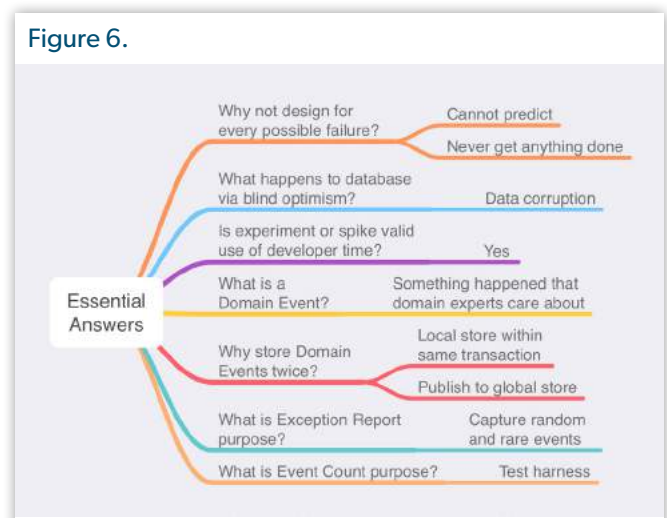
Essential Questions Answered

See Figure 6.

- *Why can't we design for every possible failure?* We can't predict every possible failure, and we'd never get anything done.
- *What can happen to a database in the path of blind optimism?* Data corruption such as missing related records.
- *Is an experiment or spike a valid use of developer time?* According to Domain-Driven Design, Scrum, Extreme Programming, and similar Agile methodologies, yes, it is.
- *What is a Domain Event in DDD?* Something happened that domain experts care about.
- *Why are we storing Domain Events twice?* We store the Domain Event in the local Event Store within the same transaction and then publish the event. Upon receipt, we store the published event in the global Event Store.
- *What is the Exception Report's purpose?* Capture random and rare events.

- *What is the Event Count's purpose?* A test harness enables us to trigger the "random and rare" code path.

Figure 6.





Summary

We recognized that “random and rare” failures are often ignored. If we came up with a contingency plan for every possible failure with every possible line of code, we’d never get anything done!

We looked at the idea of domain events and specifically at the advice that any domain event must be persisted in the same transaction as the item of interest. Then, if and only if that whole transaction was successful, publish the domain event.

PHP frameworks understand the need to save or update related data in the same transaction. All must succeed or fail as a unit; otherwise, we have an inconsistent state. However, the framework will not recognize the domain event as needing to be within the same transaction. We’ll need to begin managing transactions explicitly within our PHP code as we develop this feature.

While this appears to be more work, this insight allows us to capture information when things go wrong. We will catch and record those “rare and random” happenings that can cause problems down the line.

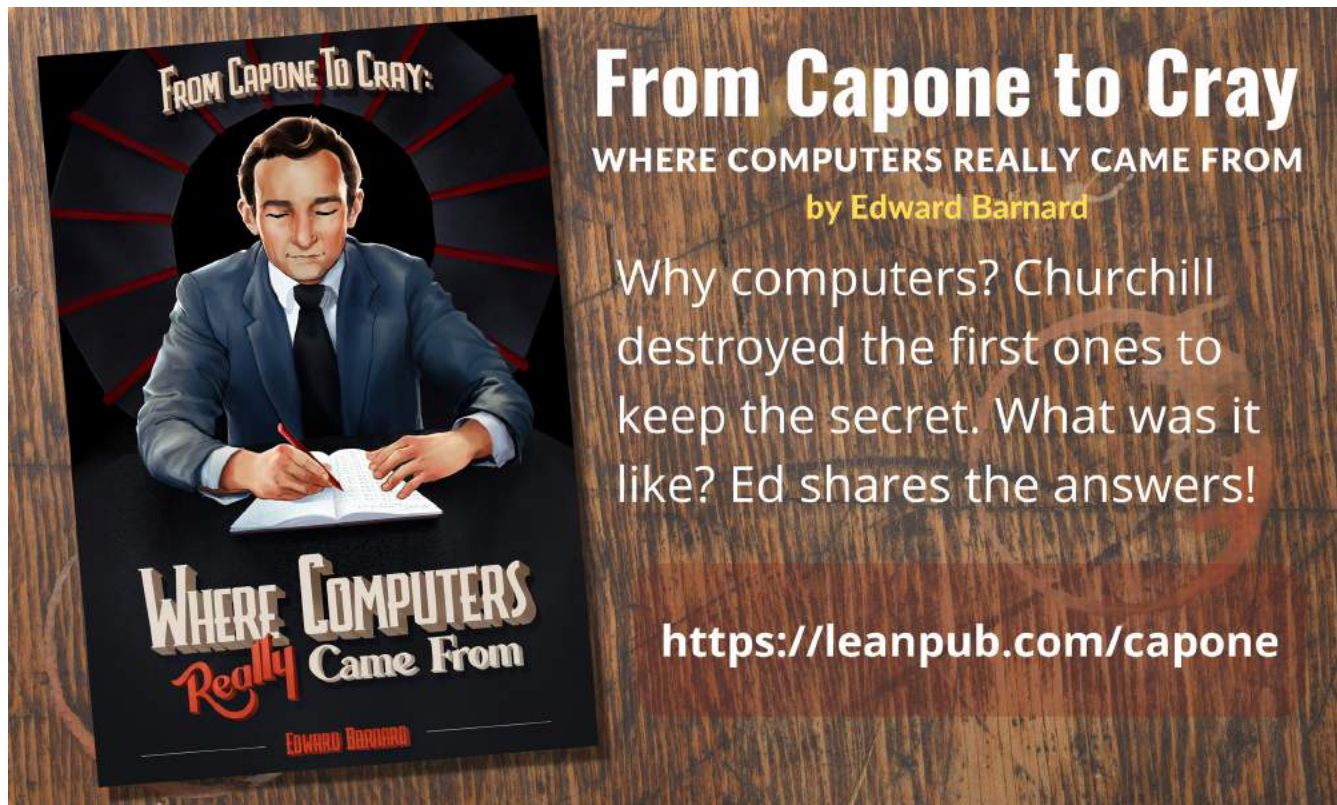
Next month, with “Structure by Use Case,” we’ll learn the Strategic Domain-Driven Design pattern that we’ll be repeating over and over as we build out our project.



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](https://twitter.com/ewbarnard)

Related Reading

- *DDD Alley: Better Late Than Never* by Edward Barnard, March 2022.
<https://phpa.me/2022-03-ddd>
- *DDD Alley: When the New Requirement Arrives* by Edward Barnard, April 2022.
<https://phpa.me/2022-04-ddd>
- *DDD Alley: Get Organized and Get Started* by Edward Barnard, May 2022.
<https://phpa.me/2022-05-ddd>



Assessing Cybersecurity Risks

Eric Mann

Every application will, one day, be exposed to a cybersecurity risk. Learning how to categorize and rate those risks is critical to keeping your team focused on the things that matter most.

I get a frequent question from fellow developers: “how do I know what to protect against.” We learn pretty quickly how to build software and how to satisfy customer needs. However, learning how to identify and adequately understand the severity of security risks is more difficult and nuanced.

In April, security researchers uncovered a very concerning vulnerability¹ in the cross-platform 7-Zip application that impacted how it operates on Windows. Specifically, an attacker could gain privileged access on Windows by dropping a specially-crafted .7z file onto the 7-Zip Help window.

💡 *There is some dispute over this particular vulnerability. It requires the local system to be configured in a specific way (i.e., with an enabled local administrator account). However, the risks presented by this vulnerability are grave enough that system administrators need to take notice.*

Privilege escalation into the administration level is one of the worst kinds of vulnerabilities in software today. Still, the project maintainers initially refused to take responsibility for the issue, a decision that left a lot of folks wondering if the vulnerability was really that big of a deal.

With any vulnerability, you should always take the time to assess the level of risk it presents to you, your application, your company, and your users. There are two vital steps required to make this judgment: categorization and risk grading.

Threat categorization

Not every threat or vulnerability is the same, so it's important to understand how each can be categorized. A helpful mnemonic for categorizing risks is STRIDE².

STRIDE breaks risks down into six categories:

- **Spoofing**—A spoofing attack raises questions about the authenticity of messages or data. Can you really be sure that information came from the source it claims? Spoofing covers any vulnerabilities or attacks related to the impersonation of trusted parties.
- **Tampering**—How likely is it that the integrity of the data you're using has been compromised?
- **Repudiation**—This is the other of a two-sided coin, including spoofing. If a piece of data alleges it came

from a certain party, can you prove that this is true? Likewise, is it impossible for said party to *disprove* that fact?

- **Information Disclosure**—Is sensitive information kept confidential?
- **Denial of Service**—How available is your application or service to legitimate users?
- **Elevation of Privilege**—Are administrative and other privileged operations properly authorized?

Once you understand the category into which a threat falls, you can gauge how critical that threat is to your organization. If there is no sensitive information in your system, “information disclosure” might not be a family of threats critical to your team. Understand these are still threats, but they merely might not be as pressing.

In the 7-Zip example, we are clearly dealing with an “elevation of privilege” threat. The question remains: just how critical is this threat for us?

Grading threat risks

While there are several different ways to rate or grade identified threats, one of the easier mechanisms³ is another mnemonic: DREAD. Like STRIDE, this is an acronym representing:

- **Damage**—What is the worst that can happen?
- **Reproducibility**—How difficult is it to reproduce an attack or exploit?
- **Exploitability**—How much technical skill is required to launch an attack?
- **Affected Users**—How many users are impacted?
- **Discoverability**—How hard is the threat to find in the first place?

For each of these dimensions, you grade a risk or a threat on a scale of 0 to 10. The meaning of each score differs based on the dimension:

- **Damage** - 0 is very trivial, 10 is full administrative control of a system
- **Reproducibility**—0 is very difficult; 10 is trivially implemented through tools like a default browser
- **Exploitability**—0 requires advanced knowledge or training; 10 is something a novice programmer could exploit

¹ vulnerability: <https://phpa.me/tomshardware-7zipzero>

² STRIDE: <https://w.wiki/Lkd>

³ mechanisms: <https://w.wiki/5CAo>



- Affected users—0 is very few users (perhaps only one); 10 is everyone
- Discoverability—0 is unlikely ever to be discovered; 10 is publicly disclosed

Once you have a score for each dimension, you average everything together, resulting in a composite risk score on a scale from 0 to 10. A composite score of 0 isn't a real risk at all. A score of 10 means you should *stop reading this article and declare an incident immediately!*

Usual risk scores will fall somewhere between the two extremes, and it's up to your team to identify the appropriate response level for each score. Generally, anything scoring a 5 or lower is likely something that can sit on the back burner. Anything 8 or higher likely warrants immediate attention from the team.

Again looking to 7-Zip as an example, we can rate the risk presented by the April exploit using DREAD:

- D: This allows administrative execution: 10
- R: It's trivially reproducible, but *only* on Windows: 5
- E: Anyone can drag an icon ...: 10
- A: Since you need direct access to a machine, this is low impact: 1
- D: It's published!: 10

Our DREAD risk rating for 7-Zip's flaw turns out to be $36/5 = 7.2$. A score representing a severe risk and their team's immediate lack of response to patch things adequately is a

sore disappointment. That said, it's also a Windows-only risk. Hence, the overall score for a Linux-only or Mac-only development team is effectively 0.

Go forth ...

Not every risk or vulnerability you face requires immediate attention. The purist in me would love to say otherwise. The reality of business is that every application and project has flaws; otherwise, we'd never ship in the first place. The trick is to understand which flaws need immediate focus and attention and which present an acceptable level of risk so that you can de-prioritize them and focus on more important activities.

Don't let the fact that risks and issues exist paralyze your team. Learn how to recognize and properly categorize the threats that exist. Then focus on quantifying the level of risk they present to your business. Armed with these two tools you can triage any bug report properly and keep your team on track.



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](https://twitter.com/EricMann)

Secure your applications against vulnerabilities exploited by attackers.

Security is an ongoing process not something to add right before your app launches. In this book, you'll learn how to write secure PHP applications from first principles. Why wait until your site is attacked or your data is breached? Prevent your exposure by being aware of the ways a malicious user might hijack your web site or API.

Order Your Copy
<http://phpa.me/security-principles>





A Night With Symfony

Joe Ferguson

Symfony¹ is one of the stalwarts of modern PHP, dating back to 2005. Symfony has continued to evolve into a framework of components focused on building web applications using decoupled and reusable components. Their philosophy of embracing professionalism, best practices, standards, and interoperability of applications is an exciting statement. Backing up that statement is years of open source community contribution and cooperation among the Symfony team, open-source contributors, partners, and enterprise developers. This month we're diving into Symfony 6 with a tour of the ecosystem and building our own demo application.

Why should you use Symfony instead of Laravel or any other framework? Thousands of developers around the world have built their careers on Symfony. When it comes to framework choice, the marketability of those skills should be part of the conversation. Symfony itself has six good reasons² you should use Symfony. Reputation, Permanence, References, Innovation, Resources, and Interoperability sound like fantastic reasons. If you are new to the PHP world or a grumpy old

developer, Symfony is a great tool to have in your tool belt. If you've spent time with Laravel, you'll find things quite similar, considering Laravel uses many Symfony components. We'll also find Screencasts³ devoted to teaching us everything we'd want to know about working with Symfony.

We're going to test drive a traditional web application using PHP 8.1 and Composer. We'll bootstrap our application with `composer create-project symfony/skeleton symfony`. Since we want some immersion in the ecosystem, we're going to install the Symfony CLI⁴. Once installed, we can verify our PHP versions and extensions have been installed with `symfony check:requirements`. You might need to install the `php8.1-intl` or `php8.1-mbstring` extensions. Check your operating system or package manager's configuration for the specific PHP extension package names. We're looking for "Your system is ready to run Symfony projects." Now we're ready to bootstrap our fresh application by running `symfony new symfony --webapp` to create a new folder named `symfony` containing our application.

We can verify our application's framework version with `php bin/console about`.

```
$ php bin/console about
-----
Symfony
-----
Version      6.0.7
Long-Term Support No
End of maintenance 01/2023 (in +281 days)
End of life      01/2023 (in +281 days)
-----
Kernel
-----
Type          App\Kernel
Environment    dev
Debug         true
Charset       UTF-8
Cache directory ./var/cache/dev (7.1 MiB)
Build directory ./var/cache/dev (7.1 MiB)
Log directory  ./var/log (0 B)
-----
PHP
-----
Version      8.1.5
Architecture 64 bits
Intl locale   en_US_POSIX
Timezone     America/Chicago (2022-04-25T12:...)
OPcache      true
APCu         false
Xdebug       false
-----
```

```
$ symfony server:start --no-tls
```

```
Tailing Web Server log file (/home/halo/.symfony5/log/
237194a723a15ec5b3688f871e8465fc4074845b.log)
Tailing PHP-CGI log file (/home/halo/.symfony5/log/
237194a723a15ec5b3688f871e8465fc4074845b/
79ca75f9e90b4126a5955a33ea6a41ec5e854698.log)
```

```
[WARNING] The local web server is optimized for local development
and MUST never be used in a production setup.
```

```
[OK] Web server listening
The Web server is using PHP CGI 8.1.5
<http://127.0.0.1:8000>
```

```
[Web Server ] Apr 25 13:03:25 |DEBUG| PHP   Reloading PHP versions
[Web Server ] Apr 25 13:03:25 |DEBUG| PHP   Using PHP version 8.1.5
(from default version in $PATH)
[Web Server ] Apr 25 13:03:25 |INFO| PHP   Listening
path="/usr/bin/php-cgi8.1" php="8.1.5" port=46363
[Web Server ] Apr 25 13:03:43 |WARN| SERVER GET (404) / ip="::1"
[Web Server ] Apr 25 13:03:43 |INFO| SERVER GET (200) /_wdt/ddc5d9
[Web Server ] Apr 25 13:03:43 |WARN| SERVER GET (404) /favicon.ico
```

1 Symfony: <https://symfony.com>

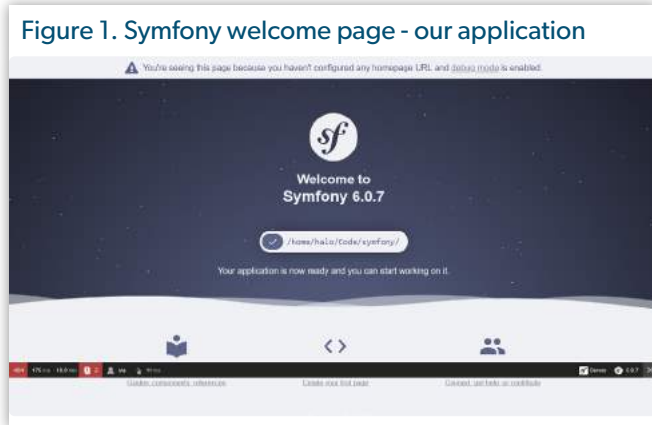
2 six good reasons: <https://symfony.com/six-good-reasons>

3 Screencasts: <https://symfonycasts.com>

4 Symfony CLI: <https://symfony.com/download>



We can launch the Symfony server to serve our application via `symfony server:start`:



Viewing `http://localhost:8000` in our browser shows us our application is ready to go!

If you prefer setting up the demo application in Docker, there's a [Symfony Docker⁵](#) guide. If you're struggling to use the Symfony command line server, you can set up the project in your local development environment.

Opening the project in your text editor of choice, we can see the project directory structure:

By default Symfony has the `DATABASE_URL` configured for PostgreSQL, I commented this line out and inserted my own value for MySQL in my `.env.local` file:

```
DATABASE_URL="mysql://homestead:secret@localhost:3306/symfony?charset=utf8mb4"
```

Remember to never put any passwords or API keys in any files that end up in version control. One way Symfony helps is by encouraging the use of `.env.local` for overriding the defaults in `.env`, and we do *not* commit `.env.local` to our repository.

Symfony uses the HTTP Request-Response, which should be common for PHP developers. It just so happens Symfony has a primer on Symfony and HTTP Fundamentals⁶, which will help illustrate what we mean by Requests and Responses.

⁵ *Symfony Docker:*
<https://symfony.com/doc/current/setup/docker.html>

⁶ *Symfony and HTTP Fundamentals:*
<https://phpa.me/symfony-http-fundamentals>

Figure 2.

```
> assets
> bin
✓ config
  > packages
  > routes
  🐘 bundles.php
  🐘 preload.php
  ! routes.yaml
  ! services.yaml
✓ migrations
  📄 .gitignore
✓ public
  🐘 index.php
✓ src
  > Controller
  > Entity
  > Repository
  🐘 Kernel.php
> templates
> translations
> var
> vendor
⚙️ .env
≡ .env.local
📄 .gitignore
{} composer.json
{} composer.lock
🐳 docker-compose.override.yml
🐳 docker-compose.yml
{} package.json
≡ symfony.lock
📦 webpack.config.js
```

Creating Our First Symfony Page

We're going to start by building a `src/Controller/IndexController.php` file containing a simple method to return a basic HTML response.

Listing 1.

```
1. <?php
2. // src/Controller/IndexController.php
3. namespace App\Controller;
4.
5. use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6. use Symfony\Component\HttpFoundation\Response;
7.
8. class IndexController extends AbstractController
9. {
10.     public function index(): Response
11.     {
12.         return new Response(
13.             '<html><body>Welcome To Symfony</body></html>'
14.         );
15.     }
16. }
```

With our `IndexController` ready to handle our route, we need to define our route in `config/routes.yaml`. Yes, YAML, I know you might hate YAML, but before you bail out because of YAML, try this *one neat trick*: turn on white space indicators in your editor!

With “Render Whitespace” off as in Figure 3.

With Render Whitespace on. See Figure 4.

Render Whitespace toggle was a game-changer for me working with YAML files. Whitespace makes it to see when things fall out of line, even in the largest and nastiest YAML file. Give it a shot and power through!

Figure 3. Render Whitespace off

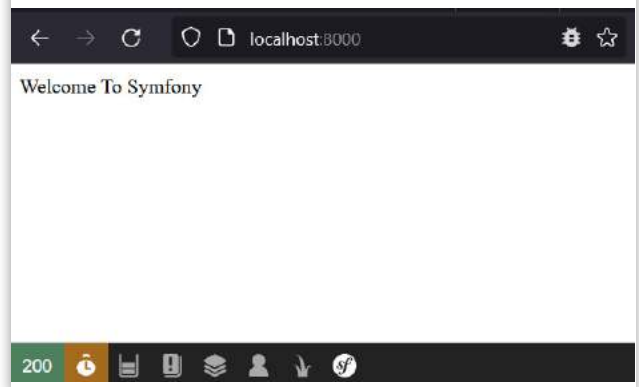
```
config > ! routes.yaml > {} kernel
    You, 23 minutes ago | 1 author (You)
1  ∨ controllers:
2      resource: ../src/Controller/
3      type: annotation
4
5  ∨ kernel:
6      resource: ../src/Kernel.php
7      type: annotation
8
```

To add our index or / route to `config/routes.yaml` we'll add the following content:

```
index:
    path: /
    controller: App\Controller\IndexController::index
```

Saving our changes and refreshing our browser takes us from the beautiful demo Symfony page to what can only be described as the pinnacle of my design skills: black text on a white background.

Figure 5.



We might be lacking in design, but that is no reason we should be returning raw HTML from a controller. Let's take a look at what Symfony offers for a webpack bundle⁷. The `webpack-encore-bundle` is similar to Laravel Mix, an opinionated way to build your application's frontend. We can use this bundle to help set our front end on a better path by running

Figure 4. Render Whitespace on

```
config > ! routes.yaml > {} kernel
    You, 23 minutes ago | 1 author (You)
1  ∨ controllers:
2      ....resource: ../src/Controller/
3      ....type: annotation
4
5  ∨ kernel:
6      ....resource: ../src/Kernel.php
7      ....type: annotation
8
```

⁷ webpack bundle: <https://phpa.me/symfony-install-encore>



composer require symfony/webpack-encore-bundle then npm install, and compile our assets by running npm run dev:

```
$ npm run dev

> @ dev /home/haLo/Code/symfony
> encore dev

Running webpack ...

DONE Compiled successfully in 871ms

5 files written to public/build
Entrypoint app [big] 531 KiB = runtime.js 14.5 KiB
... 497 KiB app.css 390 bytes app.js 18.7 KiB
webpack compiled successfully
```

We'll push our HTML response to a Twig⁸ template file located in templates/index.html.twig containing the following code to extend our base layout:

```
{% extends 'base.html.twig' %}

{% block body %}
    <h1>php[architect]</h1>
    Welcome To Symfony
{% endblock %}
```

With our HTML content removed from our src/Controller/IndexController.php, we can now return our view as a response via return \$this->render(), passing in the name of our template file.

```
public function index(): Response
{
    return $this->render('index.html.twig');
}
```

Note: *Symfony already knows templates are in the template/ folder in the application structure.*

Reloading our browser shows us a slightly better page—at least we have a nice gray background from the front-end encore package we installed. See Figure 6.

We could also have used \$this->renderView() to render a template and return a new Response() object with our contents. Symfony will handle the response information for us. We can also pass data into our views to be consumed by our templates by passing an array to our view method as shown in Listing 2.

Using composer, we can pull in packages to help get our application connected to our Relational Database Management System (RDBMS). In our examples, we'll be using MySQL, and we've already created a database earlier when we configured our DATABASE_URL configuration value.

⁸ Twig: <https://twig.symfony.com/doc/3.x/>

```
$ composer require symfony/orm-pack
$ composer require --dev symfony/maker-bundle
```

If you have not already created your database, we can create it by running php bin/console doctrine:database:create, which reads our .env and creates the database specified. If you've already created the database, you'll see an error such as "Can't create a database; the database exists." If this command completes without connection or user authorization failures, you can be confident the database configuration is ready to go. If you are receiving errors, review the documentation for Configuring the Database⁹.

At this point in a Laravel application, you would start making your migrations and building your models. With Symfony, we want to focus on our Entity classes, which loosely compare to traditional Laravel models. You'll notice Symfony's verbosity start to appear as we begin to create our first Entity class by running php bin/console make:entity to create a Widget entity as shown in Figure 7.

With our Widget entity src/Entity/Widget.php and Repository src/Repository/WidgetRepository.php classes created,

Figure 6. Symfony Encore, Webpack ready to go!



Listing 2.

```
1. public function index(): Response
2. {
3.     $data = $this->dataService->getData();
4.     $contents = $this->renderView(
5.         'product/index.html.twig', [
6.             'data' => $data
7.         ]
8.     );
9.     return new Response($contents);
10. }
```

⁹ Database: <https://phpa.me/symfony-doctrine-config-db>

Figure 7. creating a new entity class with php bin/console make:entity

```
Can this field be null in the database (nullable) (yes/no) [no]:
> no
updated: src/Entity/Widget.php
Add another property? Enter the property name (or press <return> to stop adding fields):
> stock
Field type (enter ? to see all types) [string]:
> integer
Can this field be null in the database (nullable) (yes/no) [no]:
> no
updated: src/Entity/Widget.php
Add another property? Enter the property name (or press <return> to stop adding fields):
> sellable
Field type (enter ? to see all types) [string]:
> boolean
Can this field be null in the database (nullable) (yes/no) [no]:
> no
updated: src/Entity/Widget.php
Add another property? Enter the property name (or press <return> to stop adding fields):
>
Success!
Next: When you're ready, create a migration with php bin/console make:migration
php bin/console make:entity: 02:50
```

we're ready to create our migrations. Symfony will generate our migrations for us based on our Entity classes.

Inspecting our `src/Entity/Widget.php` (Listing 3) shows that using the Symfony console `make:entity` commands saves us a lot of time and generates high-quality code based on our descriptions. Our properties are configured, and our getter and setter

```
$ php bin/console make:migration
Success!
Next: Review new migration "migrations/Version20220426173535.php"
Then: Run migration php bin/console doctrine:migrations:migrate
See <https://phpa.me/DoctrineMigrationsBundle>
```

Listing 3.

```
...
9. class Widget
10. {
11.     /**
12.      * @ORM\Id()
13.      * @ORM\GeneratedValue()
14.      * @ORM\Column(type="integer")
15.      */
16.     private $id;
17.
18.     ...
19.
20.     private $sellable;
21.
22.     public function getId(): ?int
23.     {
24.         return $this->id;
25.     }
26.
27.     public function getPrice(): ?int
28.     {
29.         return $this->price;
30.     }
31.
32.     ...
33. }
90. }
```

Listing 4.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace DoctrineMigrations;
6.
7. use Doctrine\DBAL\Schema\Schema;
8. use Doctrine\Migrations\AbstractMigration;
9.
10. /**
11.  * Auto-generated Migration: Please modify to your needs!
12.  */
13. final class Version20220426174739 extends AbstractMigration
14. {
15.     public function getDescription(): string
16.     {
17.         ...
18.     }
19.
20.     public function up(Schema $schema): void
21.     {
22.         // this up() migration is auto-generated, please
23.         // modify it to your needs
24.         $this->addSql('CREATE TABLE widget
25.             (id INT AUTO_INCREMENT NOT NULL,
26.              ...
27.              $this->addSql('CREATE TABLE messenger_messages (
28.                  id BIGINT AUTO_INCREMENT NOT NULL, body LONGTE
29.                  ...
30.              )
31.     }
32.
33.     public function down(Schema $schema): void
34.     {
35.         ...
36.     }
37. }
```

methods have already been filled out for us. If we do something wrong, we can simply rerun the commands and recreate our migrations. We can also edit the entity files directly and then use `php bin/console make:entity --regenerate` to create the getter and setter methods for the new properties.

Inspecting our migration, we can see Symfony has written out the SQL queries to build our Widget table. See Listing 4.

Once we're satisfied with our migration, we can apply the changes to our database with `doctrine:migrations:migrate`:

We can inspect our freshly created schema in the MySQL command line:

Our next step is to create a controller with a method that will create a Widget Controller class:

```
$php bin/console doctrine:migrations:migrate
```

```
WARNING! You are about to execute a migration in database
"symfony" that could result in schema changes and data loss.
Are you sure you wish to continue? (yes/no) [yes]:
> yes
```

```
[notice] Migrating up to DoctrineMigrations
[notice] finished in 66.7ms, used 22M memory, 1 migrations
executed, 2 sql queries
```




```
mysql> use symfony;
Database changed
mysql> describe widget;
+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Ex |
+-----+-----+-----+-----+-----+-----+
| id         | int       | NO   | PRI | NULL    | au |
| price      | int       | NO   |     | NULL    |   |
| description | longtext  | NO   |     | NULL    |   |
| stock      | int       | NO   |     | NULL    |   |
| sellable   | tinyint(1) | NO   |     | NULL    |   |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

```
$ php bin/console make:controller WidgetController
```

```
created: src/Controller/WidgetController.php
created: templates/widget/index.html.twig
Success!
Next: Open your new controller class and add some pages!
```

Symfony generates our `src/Controller/WidgetController.php` class (Listing 5) with the typical boilerplate and an `index()` method.

We need to go back to our `config/routes.yaml` and add a route to listen for `/widgets` and pass it to our `index()` method:

```
widget-index:
  path: /widgets
  controller: App\Controller\WidgetController::index
```

Viewing our new route, we can see Symfony also created a template for us at `templates/widget/index.html.twig`

To save a new Widget in our database we can use a `create()` method for our basic example. See Listing 6.



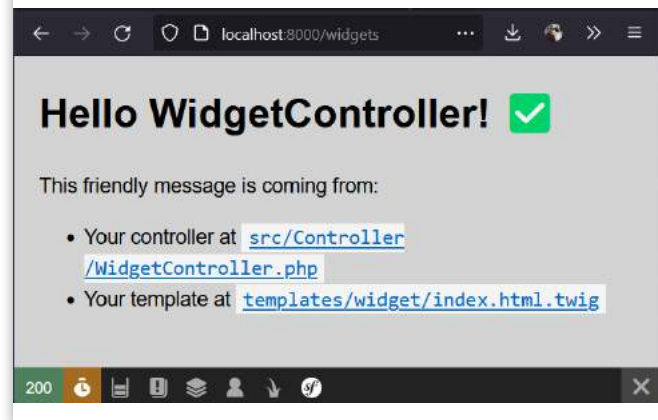
Daniel Stori {turnoff.us}

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

Listing 5.

```
1. <?php
2.
3. namespace App\Controller;
4.
5. use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6. use Symfony\Component\HttpFoundation\Response;
7. use Symfony\Component\Routing\Annotation\Route;
8.
9. class WidgetController extends AbstractController
10. {
11.     public function index(): Response
12.     {
13.         return $this->render('widget/index.html.twig', [
14.             'controller_name' => 'WidgetController',
15.         ]);
16.     }
17. }
```

Figure 8. Viewing our default widget template



Listing 6.

```
1. use App\Entity\Widget;
2. use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
3. use Doctrine\Persistence\ManagerRegistry;
4. use Symfony\Component\HttpFoundation\Response;
5. ...
6. public function createWidget(ManagerRegistry $doctrine): Response
7. {
8.     $entityManager = $doctrine->getManager();
9.
10.    $widget = new Widget();
11.    $widget->setPrice(42);
12.    $widget->setDescription('We forgot our name filed');
13.    $widget->setStock(10);
14.    $widget->setSellable(True);
15.    // We want to save our $widget, but not run the query.
16.    $entityManager->persist($widget);
17.    // flush() actually executes our insert query build
    from our $widget
18.    $entityManager->flush();
19.    return new Response('Created our Widget with ID:
    '.$widget->getId());
20. }
```



Listing 7.

```
1. public function show(ManagerRegistry $doctrine): Response
2. {
3.     $widget = $doctrine->getRepository(Widget::class)->find(1);
4.
5.     if (!$widget) {
6.         throw $this->createNotFoundException(
7.             'No widget information found for id 1'
8.         );
9.     }
10.
11.     return $this->render('widget/show.html.twig',
12.         ['widget' => $widget]);
13. }
```

With our one Widget inserted into our database with an id of 1, we can create a show method to pull our Widget from the database and pass it to a template as shown in Listing 7.

Our show() method does a look-up of our Widget's id using the find() command. If we don't get a Widget back from the query, we can throw a not found exception to alert the end-user that something went wrong. Finally, we can return a template passing our \$widget data to the template for presentation.

Thanks for coming along on our introductory Symfony night. I hope you learned something about Symfony to encourage you to take a closer look for yourself, or maybe you can see where other frameworks draw and share inspiration from each other. Ultimately the cooperation of frameworks to continue to innovate and iterate with modern changes is a positive effort for PHP developers. We've just barely scratched the surface of Symfony. There's an incredibly vibrant community waiting for you to say hello and big conferences around the world to gather with other Symfony developers.



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](#)

Using Xdebug to squash bugs, identify bottlenecks, and boost productivity?



Become a Pro or Business supporter to help ongoing development.

Supporters get help via email and elevated issue priority.

<https://xdebug.org/support>

support@xdebug.org



Clues for Hues

Oscar Merida

In the last two months, we saw how to control randomness. This month, we let users guess the random sequence created by the program.

This month's challenge builds on what we did in the last two months. From there, we have to solve some seemingly straightforward string comparisons. We'll use some less well-known PHP functions to help us build a solution.

Recap—guess the Colors

We'll build on last month's solution by writing a program that allows users to guess a random sequence of colors. When a user enters a guess, the program should indicate:

1. If the colors are in the correct place
2. If a color is part of the solution but in the wrong position
3. If a color is not part of the solution
4. If the user has guessed the correct sequence

For bonus points, allow the user to make multiple guesses until they arrive at the correct sequence. You can use whatever you want to get user input, either a webpage or the command line.

Inputs to Strings

Let's start by adjusting last month's color generator. We'll make it a function that returns a string of characters. See Listing 1. Each character represents one of the colors possible in our sequence. Since two colors start with the letter B, I used K to represent Black. Instead of working with strings though, we'll use arrays to hold our guesses and solution.

We can call it like so:

```
$answer = getAnswer(54128994);  
var_dump($answer); //string(6) "GGRPYP"
```

When a user enters a guess, we need to change their string guess into an array.

```
$guess = str_split("KYRBPP");  
var_dump($guess);
```

Checking a Guess

A loop (Listing 2) is a straightforward approach for checking a guess against our answer. Be careful with your if statements here since it's in a foreach statement. Forgetting the elseif and using a second if (like I did on my first attempt) means you overwrite the case where a guess is in the correct placement.

As a starting solution, this works; case in point:

```
ANSWER:   GGRPYP  
GUESS:    KYRBPP  
FEEDBACK: -?+-?+
```

Listing 1.

```
1. function getAnswer(int $seed) {  
2.     mt_srand($seed);  
3.  
4.     $colors = ['R', 'G', 'B', 'Y', 'P', 'K'];  
5.     $max = count($colors) - 1;  
6.     $sequence = [];  
7.  
8.     for ($i = 0; $i < 6; $i++) {  
9.         $sequence[] = $colors[rand(0, $max)];  
10.    }  
11.  
12.    return $sequence;  
13. }
```

Listing 2.

```
1. $guess = str_split("KYRBPP");  
2.  
3. // loop to check the guesses  
4. $feedback = [];  
5. foreach ($guess as $index => $letter) {  
6.     // assume its wrong  
7.     $feedback[$index] = '-'; // not correct  
8.  
9.     // is it in the right spot?  
10.    if ($answer[$index] === $letter) {  
11.        $feedback[$index] = '+'; //+ for Correct  
12.    } elseif (in_array($letter, $answer, true)) {  
13.        $feedback[$index] = '?'; // close  
14.    }  
15. }
```

However, it's naive in that if our guess was all one color, if the color is in the solution, we'll get more "in solutions, wrong spot" instances than we should.

```
ANSWER:  GGRPYP
GUESS:   PPPPPP
FEEDBACK: ???+?+
```

Character Frequency

Let's fix that. What do we need to do? We need a way to count how often each color (represented by a character) occurs in a string. Then, as we inspect the answer, we can see if we've run out of feedback for a specific characters. We can use the built-in `count_chars()`¹ function to do just that.

We're only interested in characters that occur in the answer, so we use mode 1 as the third parameter to `count_chars()`. Also, the function returns an array where the index is the ASCII value of the letter. We can use `chr()` to turn it back into a letter we can use later.

```
$raw = count_chars(implode('', $answer), 1);
$freq = [];
foreach ($raw as $chr => $count) {
    $freq[chr($chr)] = $count;
}
```

We'll have an array like the following, which tells us how many times each color occurs in our solution.

```
print_r($freq);
Array
(
    [G] => 2
    [P] => 2
    [R] => 1
    [Y] => 1
)
```

Now, we can loop through the user's guess and give better feedback. See Listing 3. When we indicate something about a letter in the answer, we can decrement the frequency count to track that we "used" it. Since we can give two kinds of feedback: "in the right spot" and "in the solution," we should prioritize giving the first kind over the second. To do so, we have to make two passes through our guess array to use up our "in the right spot" feedback first.

Our improved comparison gives the player more useful feedback about their guess.

```
ANSWER:  GGRPYP
GUESS:   PPPKPP
FEEDBACK: ?....+
```

User Input

Let's wrap it up by allowing the user to give feedback until they get the correct solution or run out of guesses. I'll use the command line to play the game, which means using `readline()`² to get user input. After each turn, we can show the remaining guesses and color previous guesses based on our feedback array using terminal colors.

Listing 3.

```
1. // count chars
2. $raw = count_chars(implode('', $answer), 1);
3. $freq = [];
4. foreach ($raw as $chr => $count) {
5.     $freq[chr($chr)] = $count;
6. }
7.
8. // assume it's wrong
9. $feedback = array_fill(0, 6, '-');
10. print_r($feedback);
11.
12. // we're checking if the letter is in the right spot
13. foreach ($feedback as $index => $state) {
14.     if ($guess[$index] == $answer[$index]) {
15.         // decrement frequency for second pass
16.         $feedback[$index] = '+';
17.         $freq[$guess[$index]]--;
18.     }
19. }
20.
21. // now check again and indicate "in the solution"
22. foreach ($feedback as $index => $state) {
23.     // only look at letters we haven't examined
24.     if ($state != '-') continue;
25.     $letter = $guess[$index];
26.
27.     if (in_array($letter, $answer) && $freq[$letter] > 0) {
28.         // only mark as in solution if we have uses left
29.         $feedback[$index] = '?';
30.         $freq[$letter]--;
31.     }
32. }
33.
34. echo "ANSWER:  " . implode('', $answer) . "\n";
35. echo "GUESS:   " . implode('', $guess) . "\n";
36. echo "FEEDBACK: " . implode('', $feedback) . "\n";
```

¹ `count_chars()`: https://php.net/count_chars

² `readline()`: <https://php.net/readline%60>



Listing 4 shows the cleaned-up code for a game prototype that takes user input and gives color-coded feedback after each guess. The random sequence changes every hour, so you and a friend could play the game at the same time and compare how you did.

Listing 4.

```
1. <?php
2.
3. function getAnswer(int $seed) {
4.     mt_srand($seed);
5.
6.     $colors = ['R', 'G', 'B', 'Y', 'P', 'K'];
7.     $max = count($colors) - 1;
8.     $sequence = [];
9.
10.    for ($i = 0; $i < 6; $i++) {
11.        $sequence[] = $colors[rand(0, $max)];
12.    }
13.
14.    return $sequence;
15. }
16.
17. function getFeedback(array $guess, array $answer) : array {
18.    // count chars
19.    $raw = count_chars(implode('', $answer), 1);
20.    $freq = [];
21.    foreach ($raw as $chr => $count) {
22.        $freq[chr($chr)] = $count;
23.    }
24.
25.    // assume it's wrong
26.    $feedback = array_fill(0, 6, '-');
27.
28.    // we're checking if the letter is in the right spot
29.    foreach ($feedback as $index => $state) {
30.        if ($guess[$index] === $answer[$index]) {
31.            // decrement frequency for second pass
32.            $feedback[$index] = '+';
33.            $freq[$guess[$index]]--;
34.        }
35.    }
36.
37.    // now check again and indicate "in the solution"
38.    foreach ($feedback as $index => $state) {
39.        // only look at letters we haven't examined
40.        if ($state !== '-') continue;
41.        $letter = $guess[$index];
42.
43.        if (in_array($letter, $answer)
44.            && $freq[$letter] > 0) {
45.            // only mark as in solution if we have uses left
46.            $feedback[$index] = '?';
47.            $freq[$letter]--;
48.        }
49.    }
50.    return $feedback;
51. }
52. function getGuess() : array
53. {
54.     $input = readline("Enter your guess: ");
55.
56.     // minor cleanup, users can enter any character but
57.     // that just wastes guesses
58.     $input = substr(strtoupper($input), 0, 6);
59.     return str_split($input);
60. }
61.
62. function outputGuessFeedback(int $num, array $g) {
63.     echo $num . ": ";
64.
65.     foreach ($g['guess'] as $idx => $letter) {
66.         switch ($g['feedback'][$idx]) {
67.             case '+': // correct, mark green
68.                 echo "\033[92m" . $letter;
69.                 break;
70.             case '?': // in the solution, mark yellow
71.                 echo "\033[93m" . $letter;
72.                 break;
73.
74.             default: // not in solution, red
75.                 echo "\033[91m" . $letter;
76.                 break;
77.         }
78.     }
79.     // reset color
80.     echo "\033[0m\n";
81. }
82.
83. $now = new \DateTime("now", new DateTimeZone("UTC"));
84. $answer = getAnswer($now->format('YmdH'));
85. $maxGuesses = 6;
86. $guesses = [];
87. $solved = false;
88.
89. do {
90.     $guessesLeft = $maxGuesses - count($guesses);
91.     echo "Guess the correct six-color sequence. \n";
92.     echo "R=Red, G=Green, B=Blue, Y=Yellow, P=Purple,
93.         K=Black\n";
94.     echo "\n";
95.     echo "You have $guessesLeft guesses left.\n";
96.     $guess = getGuess();
97.     $guesses[] = ['guess' => $guess,
98.                  'feedback' => getFeedback($guess, $answer)];
99.     foreach ($guesses as $i => $oldGuess) {
100.        outputGuessFeedback($i, $oldGuess);
101.    }
102.    // did they win?
103.    if ($guess === $answer) {
104.        $solved = true;
105.    }
106. } while (!$solved && count($guesses) <= $maxGuesses);
107.
108. if ($solved) {
109.     echo "CORRECT! You found the solution!\n";
110. } else {
111.     echo "You ran out of guesses.";
```


Birthday Problem

For next month's challenge, assume birth dates are distributed across the calendar year with the same probability. Write a program that calculates the probability of two people sharing a birthday given there are N people in a group. What value of N brings that probability to 50%?

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 PsySH³ 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.

³ PsySH: <https://psysh.org>



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](https://twitter.com/omerida)

Related Reading

- PHP Puzzles: Finding Integer Factors by Oscar Merida, February 2022.
<http://phpa.me/2022-02-puzzle>
- PHP Puzzles: Finding Prime Factors by Oscar Merida, March 2022.
<https://phpa.me/2022-03-puzzle>
- PHP Puzzles: Making Some Change by Oscar Merida, April 2022.
<https://phpa.me/2022-04-puzzle>
- PHP Puzzles: Controlled Randomness by Oscar Merida, May 2022.
<https://phpa.me/2022-05-puzzle>

Tackle Any Coding Challenge With Confidence

This book teaches the skills and mental processes these challenges target. You won't just learn "how to learn," you'll learn how to think like a computer.

Order Your Copy
<http://phpa.me/fizzbuzz-book>





Pulled From All Angles

Beth Tucker Long

Everywhere I look, there are articles targeting programmers with suggestions and plans for how to be “better.” Some articles promise to make you a better co-worker, while others extol the virtues of being a better manager. Some will make you better at communicating while others will make you better at coding. In all cases, though, the message is the same: you programmers need to be “better.”

I like to read work-related articles online, and I am always looking for ways to improve, but lately, I have been a bit overwhelmed by the articles I have seen promoted and shared. I went through my weekly newsletters with article suggestions as well as checked out a few sites that I frequent, and the lineup of recommended articles was contradictory and demoralizing. The first article I checked out was touted as a way to become a better overall programmer. I was hoping for some strategies to better manage projects or architect systems. Instead, it was all about how programmers are too focused on being right rather than the needs of their customers/users. The article wrapped up by explaining how we need to put our egos aside and learn some compassion.

The next article was also about becoming a better programmer, but this one criticized programmers for giving in to customer whims and not enforcing proper security protocols and architecture best practices. The takeaway from this article was that programmers need to stand up for what’s right no matter what customers/users are requesting.

Another article said that programmers need to step up and be managers because it is essential for managers to understand their team’s job. While the next one said that programmers should never be managers because the industry is so short on programmers that we can’t afford to have anyone leave their programming job.

This trend continued. Programmers need to have more face-to-face meetings to better understand their users and stay on schedule. Yet another said programmers should also stop having face-to-face meetings and handle more by email because meetings waste programming time and interrupt essential thought processes. Programmers must have a coding standard and never change it, but programmers also need to

be flexible and ignore non-critical standards to get things out the door more quickly. Programmers should never use a framework because it is just a crutch and is too bloated, but programmers should also always use a framework because it wastes time to code things that have already been coded.

So I don’t know who needs to hear this today, but you are doing a good job. Being a programmer is hard, and there is very clearly no one right answer for how to do things. All we can do is to keep learning, take all advice with a heavy dose of skepticism, and above all else, protect our mental health. Don’t let the news and blogs get you down. Don’t let your job overtake your personal life. Don’t let exhaustion lead you to burnout. All the frameworks, best practices, meeting techniques, and everything else is not worth losing yourself over. You are important.



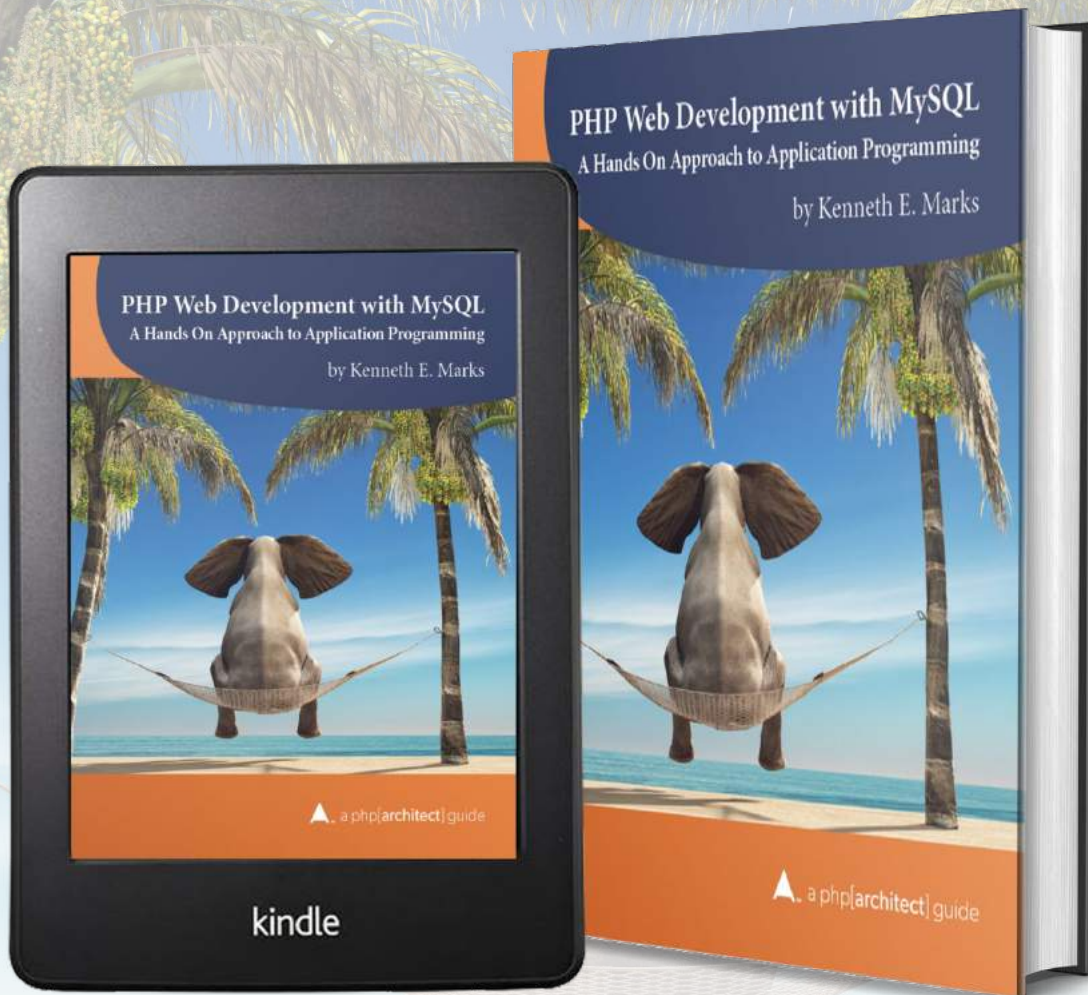
Beth Tucker Long is a developer and owner at Treeline Design¹, a web development company, and runs Exploricon², a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development³ and Full Stack Madison⁴ user groups. You can find her on her blog (<http://www.alittleofboth.com>) or on Twitter @e3BethT

1 Treeline Design: <http://www.treelinedesign.com>

2 Exploricon: <http://www.exploricon.com>

3 Madison Web Design & Development: <http://madwebdev.com>

4 Full Stack Madison: <http://www.fullstackmadison.com>



Learn how to build dynamic and secure websites.

The book also walks you through building a typical Create-Read-Update-Delete (CRUD) application. Along the way, you'll get solid, practical advice on how to add authentication, handle file uploads, safely store passwords, application security, and more.

Available in Print+Digital and Digital Editions.

Purchase Your Copy
<https://phpa.me/php-development-book>

Web Apps • Mobile Apps • E-Commerce



A DIFFERENT DEVELOPMENT EXPERIENCE

Developers who care about the code
they create, the communities they build,
and the solutions they implement.



DIEGODEV.COM

(406) PHP-CODE or (406) 747-2633