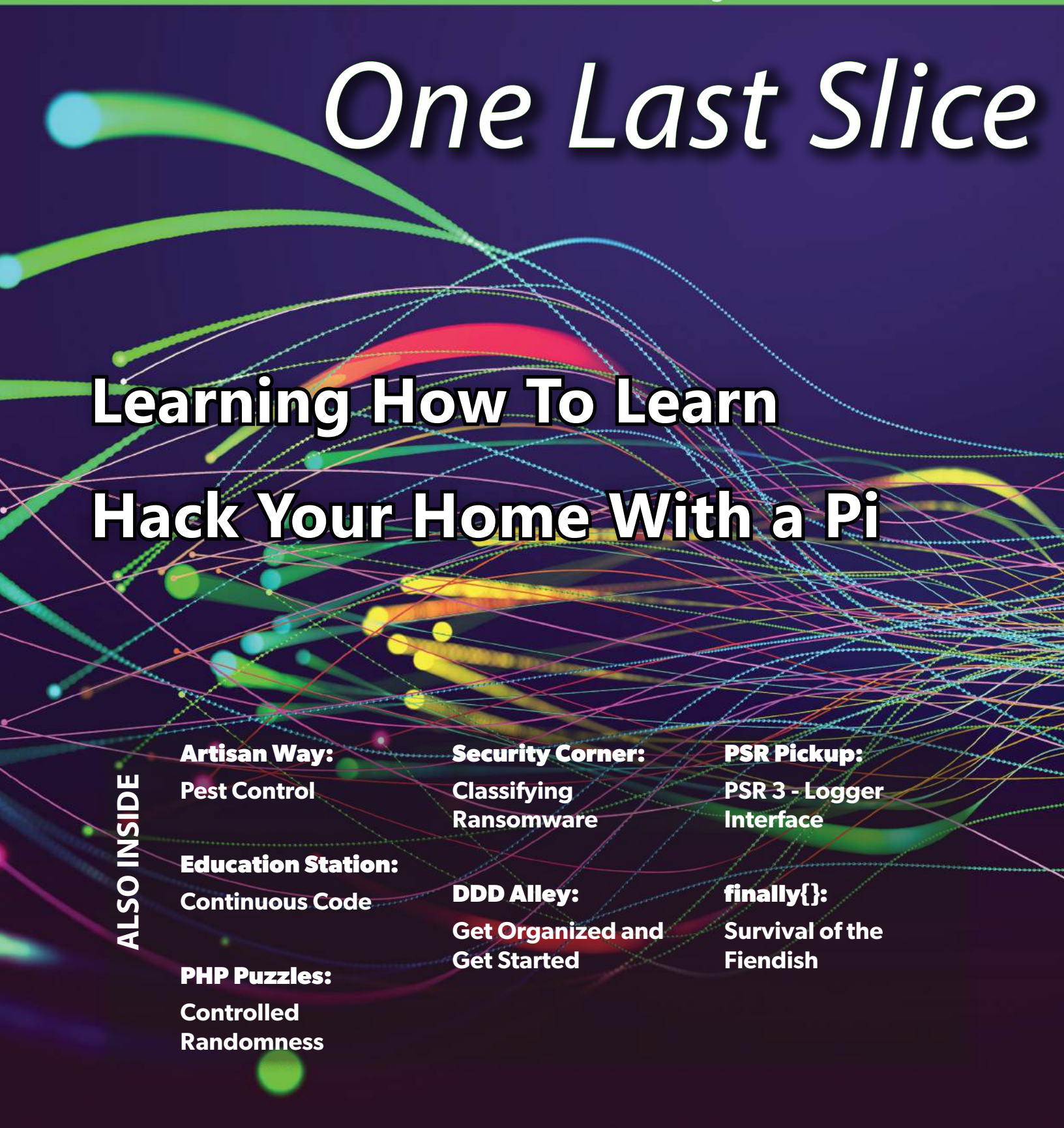
www.phparch.com

May2022
Volume 21 - Issue 5

php[architect]

The Magazine For PHP Professionals



One Last Slice

Learning How To Learn

Hack Your Home With a Pi

ALSO INSIDE

Artisan Way:
Pest Control

Education Station:
Continuous Code

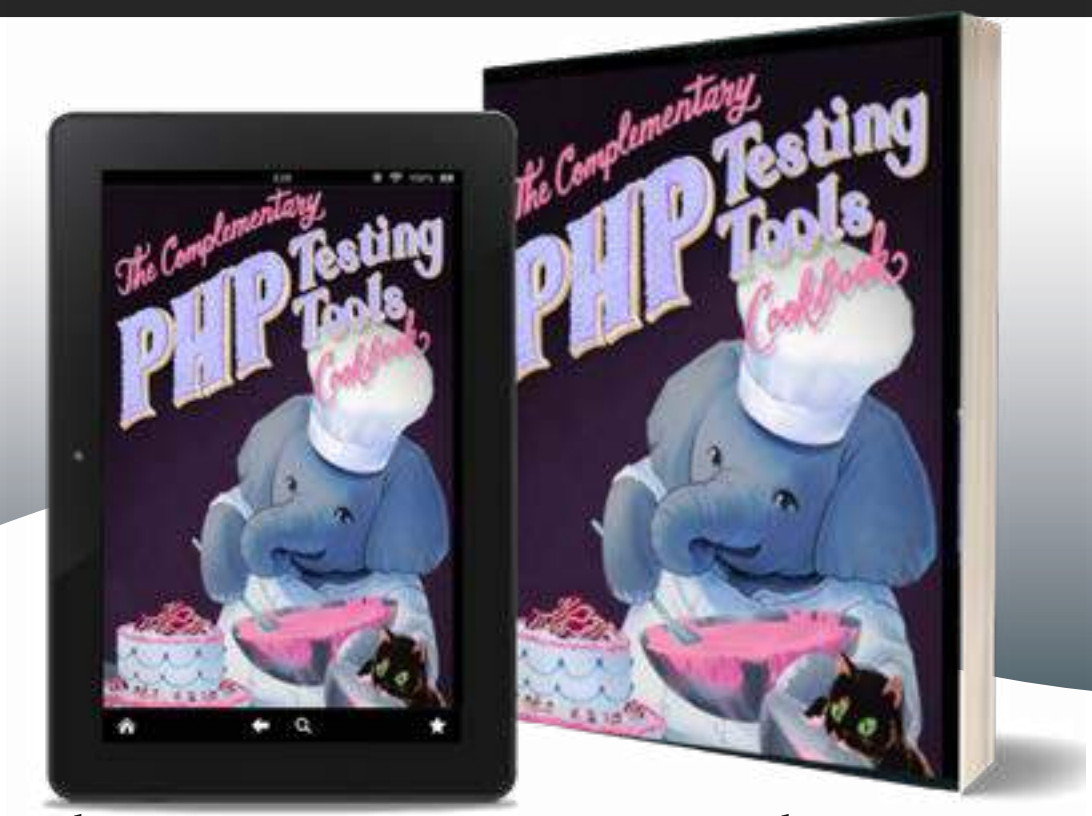
PHP Puzzles:
Controlled Randomness

Security Corner:
Classifying Ransomware

DDD Alley:
Get Organized and Get Started

PSR Pickup:
PSR 3 - Logger Interface

finally{}:
Survival of the Fiendish



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

CONTENTS

MAY 2022
Volume 21 - Issue 5

2 **A Good Problem to Have**
Eric Van Johnson

3 **Learning How to Learn**
Joel Clermont

6 **Hack your Home with a Raspberry Pi - One Last Slice**
Ken Marks

21 **Continuous Code**
Education Station
Chris Tankersley

25 **Classifying Ransomware**
Security Corner
Eric Mann

28 **New and Noteworthy**

29 **Get Organized and Get Started**
DDD Alley
Edward Barnard

37 **Pest Control**
The Artisan Way
Marian Pop

40 **Psr-3 Logger Interface**
PSR Pickup
Frank Wallen

42 **Controlled Randomness**
PHP Puzzles
Oscar Merida

44 **Survival of the Fiendish**
finally{}
Beth Tucker Long

Edited with cake (Happy Birthday Eric)

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

A Good Problem to Have

Eric Van Johnson

I thank my lucky stars that I am not a superstitious person, knock on wood, and I really don't want to jinx it, but we have a good problem.

We have so many great feature articles in the pipeline, and I think it's important to let everyone know what an awesome job the PHP community has been doing at contributing wonderful content. We even have contributors messaging us asking, "Was there something wrong with my article?" The answer is always the same, "No, I am really excited to publish it; I just need to find the room." We've even had to bump articles during the layout process because we ran out of room. This is an example of a good problem to have.

If you've ever considered writing for php[architect], now is one of the best times to start that process. There is plenty of time to work on a feature article without the pressure of a deadline looming over your head anytime soon. Be sure to check out our website[1] to see how to contribute yourself. We make the process as fun as we can, are here to help you every step of the way, and make it as rewarding as possible. Adding "Published Author" always looks good on your LinkedIn profile.

Speaking of great articles, Ken Marks gives us One Last Slice of Pi to wrap up his series on working with a Raspberry Pi. Ken has done such a fantastic job taking us through his process of using a Raspberry Pi and writing code on that Pi in a real-life scenario at home. In this final installment, Ken establishes a workflow for sending an accelerometer text message using SendMail.

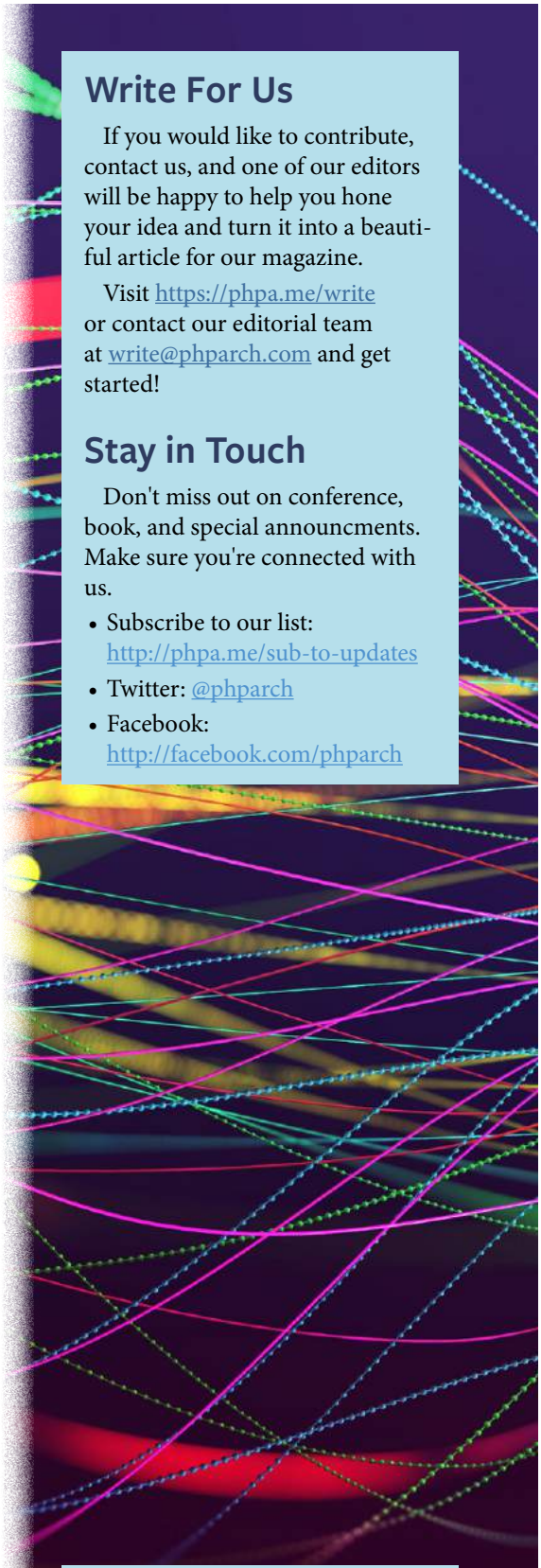
Also, this month, Joe Clermont contributed his article, "*Learning How to Learn.*" In an industry where there is a real need to always learn, Joe shares some techniques to help us with continual learning using technics such as learning by doing and learning by sharing.

In Chris Tankersly's column Education Station he discusses "*Continuous Code,*" where he discusses continuous code, automated testing, continuous integration, and continuous delivery. PSR Pickup's Frank Wallen shares an article on "*PSR-3 Logging Interface*" and talks about different log levels and messages. In Eric Mann's Security Corner, we learn about "*Classifying Ransomware*" and the differences between them.

May brings a new recurring column by Marian Pop focused on the Laravel Framework and Ecosystem. This month he introduces us to "*Pest Control*" and discusses the PEST testing package. Edward Bernard's DDD Alley gets hands-on with "*Get Organized and Get Started*" and shares his thoughts on some of the steps to getting a project set up with code first and implementing a strategic domain-driven design.

Oscar Merida adds some "*Controlled Randomness*" in this month's PHP Puzzles. It's fun for the whole family, provided your entire family are passionate developers. Lastly, in finally{}, Beth Tucker-Long talks about "*Survival of the Fiendish*" and shares her experience of trying to address a simple problem that grows more complex around every turn. That would be an example of a bad problem to have.

1 *Write For Us: https://phpa.me/write*



Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

Stay in Touch

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

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

Download the Code Archive:

http://phpa.me/May2022_code

Learning How to Learn

Joel Clermont

Software developers are, by necessity, lifelong learners. We are constantly facing new challenges, new technologies, new methods of solving problems. But how can we make our learning more efficient? How can we better retain and apply the new things we learn?

Technology is constantly changing. It could be a new version of PHP or your favorite framework. Or maybe you're trying to pick up a new skill related to coding, such as deployment, monitoring, performance optimization, debugging, or one of the dozens of other things that may be required to advance your career. This constant stream of topics to learn is one of the things I enjoy about software development. You can learn a wide breadth of skills or go as deep as you want in a limited area and never get bored.

At the same time, it can also get a bit overwhelming. In addition to the sheer volume of possible things to learn, many developers also have this nagging feeling that they haven't fully mastered the skills they've already learned. Sure, we might acknowledge competency, and we know how to get our work done, but we don't quite feel like we've achieved true mastery of a particular skill.

How do we cope with this? Well, one way is something you're already doing right now: seeking out sources of reliable information to read. It could be anything from an entire book on a topic, a blog post, a magazine article, or even a short 🔥hot tip code tweet in our feed. Another common approach is to go to a local dev meetup or a larger conference and watch someone speak on a topic. They talk enthusiastically and maybe show an amazing live demo, and we're all fired up to apply what we learn.

All these approaches can be informative and even entertaining, but often it's hard to take what we've learned and move beyond the basics. Or perhaps we feel like this new skill washes right over us, and it's difficult for us to retain what we've listened to or read. If you can relate to any of this, you are not alone! I've felt the same thing time and time again, and after thinking about this problem and talking to many other developers, I realized that **many developers** feel this same way.

The goal of this article is to share what I've learned from years of researching this topic, trying things out in my own learning experiments, and seeing what works and what doesn't. I'll share a number of tactical tips that will help you get a deeper understanding of your topic of study, and how to move from inexperience to competence to mastery. Where relevant, I'll share some interesting research about how our brain works and why some of these techniques are so effective.

Learn by doing

I've divided up these techniques into two main groups. First, let's discuss techniques that help you take a more active role in your individual learning.

Don't be afraid to talk to yourself

Reading a book can be a very passive experience if we let it. Instead of just reading page after page of information, it's helpful to slow down and more actively participate in what you're reading. For example, if the author makes a statement, "It is better to use method X instead of method Y in this circumstance," this is an opportunity to engage your brain. Before moving on, ask yourself these questions:

- Why is method X better?
- What would happen if I used method Y instead?
- Is method X always better, or is there some context that makes this statement true?

It may seem silly, but there are benefits to actually vocalizing these questions and talking through the answers. Research has shown that speaking is a more "hard-wired" ability in the human brain, whereas reading and writing are acquired skills. Engaging the speaking centers of our brain can help with understanding and retention, even if it might make someone else in the room give us a strange look.

> *You can try this technique out while reading the rest of the article. I won't hear you, but go ahead and ask some questions. Challenge my statements. Talk it out.*

One of my favorite programming books, The Little Schemer[1], takes this concept to its logical extreme. The whole book is presented in the form of an imaginary conversation where you are one of the active participants. Even if you never plan on writing any code in Scheme, I'd recommend reading it as a way to get really good at the skill of active learning.

Hands-on experimentation

In addition to getting involved mentally, it's very useful to get our hands on a keyboard and try out some new things we've learned. If the article we're reading has a code example, type it in and try it out. Don't copy and paste it either. Type the code in yourself. It may seem tedious and unnecessary,

1 *The Little Schemer:* https://mitpress.mit.edu/books/little-schemer-fourth-edition

but you're taking a more active role by typing it in. Not to mention, you may make a mistake, and then you'll get to exercise another important skill: figuring out why it isn't working.

If the book you're reading has exercises, do them. None of us likely enjoyed homework when we were in school, but there is evidence that trying something out in a guided way helps us reinforce what we've learned. What if there are no exercises provided? Make your own! Take a small piece of what was explained and see if you can apply it. Doing so will take a little bit of creativity, but the effort is worth it.

For example, I was working through a tutorial showing how to use Websockets in Golang. The tutorial code was a chat app where everything you typed in generated an automatic response from the server sending back what you typed in upper case. I made my own exercise to allow two different web users to chat with each other over that websocket connection. It took me a few hours of figuring things out, but in the end, I felt like I really understood it better. If you need some inspiration for practice exercises, I recommend the book Exercises for Programmers[2]. It contains almost 60 general exercises you can apply when developing a new coding skill.

One series of courses that has these principles baked in is Learn Code the Hard Way[3]. So often, we try to take shortcuts to make learning easier, but research on skill acquisition shows that some seemingly "hard" work upfront gives better results in the end.

Make it your own

Doing the assigned homework is useful, but to really grasp a concept, you need to fully make it your own. Can you take one of the exercises and stretch beyond its initial scope, perhaps including elements even more relevant to your day-to-day work or business domain?

One important thing to recognize is that stretching slightly beyond exactly what was presented might be a little frustrating. You might try several things, none of which work the way you expected, and end up feeling stuck. But this is actually a very good thing! Working through a little uncertainty and frustration will help cement the concept in your mind once you break through and figure it out.

There's an interesting approach to teaching called Inquiry-Based Learning, often used in teaching mathematics, where the teacher presents problems without fully explaining how to solve them. The teacher guides the class discussion, but the students work through a number of ideas before "discovering" the correct solution. It might seem inefficient to let a class struggle for a day or two on a problem when the teacher could have explained it in 15 minutes, but the evidence shows that this period of frustration followed by discovery leads to a much deeper understanding of the core concepts.

One important thing to note: there is such a thing as too much frustration. If you're stuck for days on end and don't feel like you are making any progress, it's completely acceptable to make a note, move on to the next topic, and come back to it later. Sometimes the little bit of information you need to get unstuck is in the very next chapter of the book you're studying.

As programmers, knowing how to solve a particular problem is useful, but understanding the underlying concepts unlocks a much more useful application of what we've learned. To use a cooking analogy, we move beyond following simple recipes and start creating interesting meals of our own.

Practice, practice, practice

When someone is learning to play a musical instrument, practice and repetition are absolutely critical to get better. And while it might be fun to practice a song we enjoy, quite often, time is dedicated to practicing "boring" things like musical scales. Even if you've played an instrument for many years, practicing scales is still common. Why? There are a few benefits:

- Building up muscle memory
- Improving your technique
- Understanding music theory leads to better improvisation and composition
- Training your ear—think in music phrases, less in individual notes

How can we apply this to programming? Practice the basics! String manipulation, handling user input, interacting with a database, basic data structures: are all like practicing musical scales. Don't overlook them. Code katas are small exercises meant to be repeated over and over again. You can find quite a few examples online, for example codekata.com[4] or exercism.org[5].

One related tip: Spend some time practicing both with and without your everyday tools. For example, if you're a Vim or PHPStorm pro, try a session just using Visual Studio Code without a bunch of helper tooling. Again, this will be frustrating, but you'll benefit from the experience of not having things auto-completed for you. Certainly, it makes sense to use your most productive environment when you're on the job, but take some time while learning to practice without these tools on occasion as well.

Test yourself

Every time you recall a fact, this forces your brain to strengthen that mental connection. Another way of saying it is that every **read** in your brain is also a **write**. You can leverage this to make something more permanent in your memory. When learning a new skill, it's useful to create "flashcards" of key things you want to remember. A day or two later, go through those flashcards and see how much you can recall. As you keep learning new things, keep cycling back to those older flashcards, and see if you can continue to recall them.

2 *Exercises for Programmers:* https://pragprog.com/titles/bhwb/exercises-for-programmers/

3 *Learn Code the Hard Way:* https://learncodethehardway.org

4 *codekata.com:* http://codekata.com

5 *exercism.org:* https://exercism.org

This technique is called spaced interval repetition, and there are a number of tools that help you capture this information and allow you to practice recall. Anki[6] is one system I've used that has been very helpful for practicing this technique.

Learn by sharing

The previous set of techniques focused on things you can do on your own. This next section suggests some additional activities you can try with other people to make learning more effective.

Teach someone else

For over 10 years, I was involved in running the Milwaukee PHP user group. It never ceased to amaze me how presenting something to a group, even something I knew quite well, always forced me to learn the material better. There's something about putting yourself in front of a small group, with the possibility of unexpected questions, that really makes you question how well you know a topic.

While this can be uncomfortable, having to explain to someone else how something works is an excellent way of making sure you truly understand it. You quickly identify facts you take for granted and make sure you can defend why something is true. And if you want to try this on "hard mode," try explaining it to someone outside your field. Can you explain to your cousin, the electrician, why a `use` clause is important when defining an anonymous function that needs access to variables from the parent scope?

You can get the benefits of sharing what you know even without public speaking. For example, you could write up a blog post or even submit an idea for an article to php[architect][7]. The principles are the same: sharing something requires you to understand it in a deeper way.

Pair programming

Pair programming is often held up as a beneficial practice for creating higher quality code and sharing knowledge within a team. It's also a fantastic technique for learning something new. It combines elements of both learning by doing and learning by sharing. Even better, it works whether you are pairing with someone who knows the skill well or someone who is also a beginner like you.

When pairing with a fellow beginner, you can actively discuss it with them. Instead of talking to yourself, you can talk it through with another actual human. They may have a different set of questions when approaching a new subject. They might have a different solution in mind to an exercise. Exploring this will round out your own understanding. Plus, pairing brings an element of focus that is easier to lose when working alone.

When pairing with someone knowledgeable on the skill you're learning, you get a different set of benefits. Now you have someone that can nudge you toward a solution or help you when you get stuck. In addition, they can help you understand the idiomatic way of solving something. Often there are multiple ways to solve a particular problem, but often a programming community has a more commonly accepted way of doing so. Writing code in that way will allow you to work more easily on a team with other skilled people.

Next steps

One important decision we haven't touched on yet is *what* you should learn. This can be a bit subjective, and it all depends on your goals, but I have some general advice to share:

Make it fun. Much of what we might want to learn is to help us do better at work or get a raise, but learning should be fun too. In addition to learning practical things you can use at your day job, make sure to mix in some topics that you may never use. It will broaden your horizons, keep you engaged with learning, and you might be surprised how seemingly unrelated things will prove useful later.

Try something new. If you are reading this magazine, you probably spend a fair amount of time writing PHP code. There are benefits to learning something very foreign to what you're used to learning. Instead of dabbling in another language conceptually similar to PHP (like Python, Ruby, etc.), pick something entirely different like Elm or Scheme. Like the previous point, you may not find it useful at work, but you will pick up new ways of solving problems.

Keep scope small. Decide upfront what you want to learn and limit the scope. Having a specific, achievable goal will help you stay motivated and result in satisfaction at having accomplished what you wanted. For example, maybe you want to build the canonical example of a blog to learn some new language or framework. Limit the initial scope to posting and viewing blog articles, and don't feel the need to build a fully-featured comment system. Without predefined limits, you will likely lose interest and focus over time.

Now, go forth and try some of these techniques. Pick one of the other articles from this magazine and talk to yourself as you read it, try the monthly puzzle as an exercise, type in the sample code, take it a step beyond the example, and don't be afraid to feel stuck for a day or two. Most of all, have fun along the way. Our brains don't perform well under stress, so keep an open mind and view even your failures as progress in the right direction. Finally, share what you learn. Not only will this help you, but you will "pay it forward" and benefit the community as a whole.

Joel works with Laravel teams to help them level up their skills and ship better products. Co-host of the No Compromises podcast[8]. Author of Mastering Laravel Validation Rules[9]. Organizer of Milwaukee PHP. @jclermont

6 *Anki:* https://apps.ankiweb.net

7 *php[architect]:* https://phpa.me/write

8 *No Compromises podcast:* https://show.nocompromises.io

9 *Rules:* https://masteringlaravel.io/laravel-validation-book

How to Hack your Home with a Raspberry Pi - Part 5 - Sending an Accelerometer Text Message Using SendMail

Ken Marks

Welcome back to the final installment of How to Hack your Home with a Raspberry Pi. At the end of this article, your Raspberry Pi will be able to send you a text message, letting you know that your clothes dryer cycle has finished. You will need to start from the beginning of this series if you want to build this project yourself.

Introduction

Here's what we've covered so far in parts 1 - 4:

- Part 1:
  - My motivation for this project
  - Components needed
  - Installing the OS on the Pi
  - Configuring the Pi
- Part 2:
  - Updating the software packages on your Pi
  - Installing and testing the LAMP Stack
- Part 3:
  - Connecting the accelerometer
  - Creating the database for storing accelerometer data
  - Compiling a C/C++ program that reads accelerometer data and logs it to the database
  - Creating a Unix Service to automatically start and stop logging
- Part 4:
  - Build a web service that uses the logged accelerometer data
  - Build an application that plots the accelerometer data

In this installment, we will:

- install and configure `postfix` to use Gmail SMTP
- install a State Machine that detects when our accelerometer stops shaking to detect when our clothes dryer is done
- send a text message to ourselves using `sendmail` when our drying cycle is complete

This is the part of the series where we put it all together. When I set my Raspberry Pi (with accelerometer attached) on top of my clothes dryer, I want to be notified when the drying cycle is complete. That way, I can retrieve the clothes and fold them before they get wrinkled—or turn on a wrinkle guard touch-up cycle if I'm too lazy. The easiest way I can think of to accomplish this is to send a text message to my phone through an email message to my Gmail account using SMTP. To avoid false positives about when the clothes dryer is really done drying the clothes, I will install a State Machine that will ensure the text message is only sent when the clothes dryer cycle is complete. I'll explain more about how the State Machine works below.

Power up your Pi and Remotely Log In

Ensure your accelerometer is properly connected to your Raspberry Pi (see part 3), connect the power supply to your Pi, and plug it into A/C mains.

SSH into your Pi

After a few minutes of letting the Pi power up, bring up a terminal window on your local computer. Next, SSH into your Pi with the user `pi` by typing the following command into the terminal:

```
ssh pi@raspberrypi.local
```

Enter your password. You should now be logged in to your Raspberry Pi.

Update the Packages

I'll first check to see if any software packages need upgrading on the Pi using the `apt` package manager and run the following two commands in the terminal window:

```
sudo apt update
sudo apt upgrade
```

Install and Configure postfix to use Gmail SMTP

As I mentioned above, we will need a way to notify ourselves when our clothes dryer cycle is complete. The easiest way to do that is to send a text message to our phone using `sendmail` after configuring our Gmail account to proxy SMTP messages. We will need to install the following three packages:

Package	Description
postfix	A mail transfer agent that supports the simple mail transfer protocol (SMTP) so we can relay a mail message through our Gmail account
mailutils	Contains utility functions we can use to send mail and will make it easier to send a mail message from our State Machine code.
libsasl2-modules	Modules to support the sending of mail messages using the simple authentication and security layer (SASL), which we will need for our SMTP relay through our Gmail account to work

> 💡 *NOTE: The following instructions were adapted from an article written by Stacy Prowell[1] on 12/11/2019 on how to set up Gmail and email on a Raspberry Pi.*

In the terminal window, enter the following command to install these packages:

```
sudo apt install postfix mailutils libsasl2-modules
```

You will be asked: Do you want to continue? [Y/n]. Either press `Y` and the return key or just press the return key. You will see the details of the installation process in your terminal window.

During the installation, you will be asked about the mail server configuration. Select `Internet site` configuration, tab over to <OK>, and press the return key.

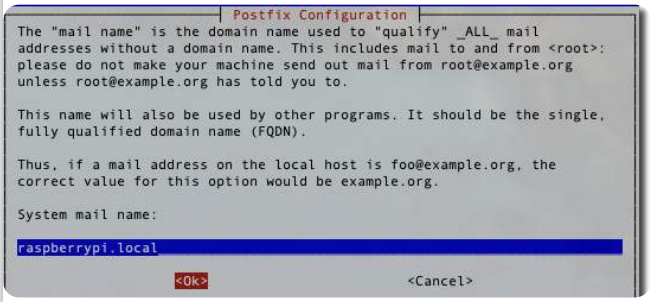
Next, you will be asked for a `System mail name`. The default name should already be set to `raspberrypi` since that is our hostname. Change it to `raspberrypi.local`, tab over to <OK>, and press the return key as shown in Figure 1.

Setup your Gmail Account to use an Application Password

> 💡 *NOTE: I am assuming you have a* Gmail *account. If you don't, they are easy enough to set up. Additionally, we will be using an Application Password to use our Gmail account as an SMTP proxy for sending email messages, a more secure mechanism than exposing your Gmail password. However, you are required to enable two-factor authentication (2FA) on your Gmail account to use Application Passwords.*

1 *Stacy Prowell:* https://phpa.me/medium-setup-mail-pi

Figure 1. postfix: system mail name



We need to generate an Application Password from our Gmail account for our Raspberry Pi to send SMTP mail messages using our Gmail account as a relay to email messages. First, we need to enable two-factor authentication on our Google account.

Open up a tab in your browser on your local computer and navigate to `https://myaccount.google.com`, log in, and select Security.

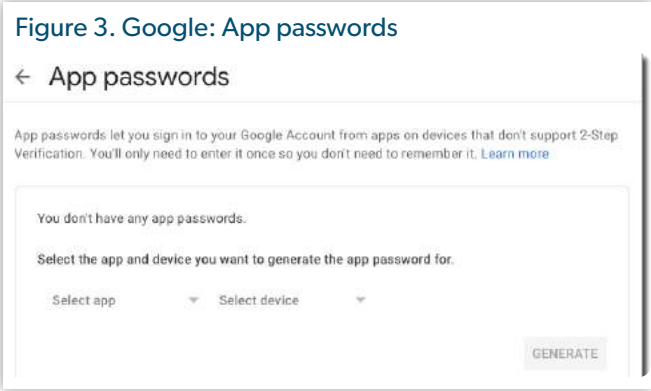
Scroll down to the **Signing into Google** section and turn on **2-Step Verification** if it is not already enabled as shown in Figure 2.

> 💡 *NOTE: If 2-Step Verification is not on, select the arrow to the right (>) and walk through the steps to configure it. You will be asked for your Google password o verify it is you.*

Figure 2. Google: 2-Step Verification - On

Now let's generate an Application Password by selecting the arrow to the right (>) of App passwords. After entering your Google password, you will be presented with the App passwords page (Figure 3).

Figure 3. Google: App passwords

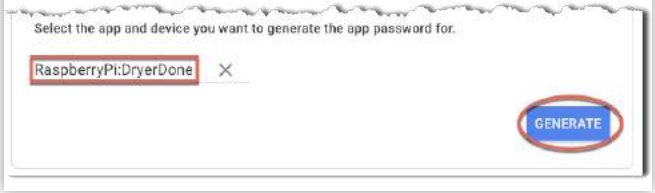


Under the Select app dropdown, select Mail. See Figure 4.

Figure 4. Google - App passwords: Select App - Mail

Under Select device, select Other, and enter: `RaspberryPi:DryerDone`. It doesn't matter what we name the device, but this let's easily identify what the Application Password is being used for. See Figure 5.

Figure 5. Google - App passwords: Select Device - RaspberryPi:DryerDone



Click the GENERATE button, and you will see a dialog with the generated application password. Copy the password or take note of it because this is the last time you will see it, and if you lose it, you will need to generate a new one.

Select Done. You will now see a list of your generated application passwords. You can delete them and generate new ones.

Listing 1.

```
1. # Enable authentication using SASL.
2. smtp_sasl_auth_enable = yes
3. # Use transport layer security (TLS) encryption.
4. smtp_tls_security_level = encrypt
5. # Do not allow anonymous authentication.
6. smtp_sasl_security_options = noanonymous
7. # Specify where to find the login information.
8. smtp_sasl_password_maps = hash:/etc/postfix/sasl/sasl_passwd
9. # Where to find the certificate authority (CA) certificates.
10. smtp_tls_CAfile = /etc/ssl/certs/ca-certificates.crt
```

Configure SASL on your Raspberry Pi

Back on our Raspberry Pi, we need to add the application password we just generated to the Simple Authentication and Security Layer (SASL) configuration.

In the terminal window where you are logged into your Pi, let's create a simple authentication and security layer (sasl) password file in the `/etc/postfix/sasl` folder called `sasl_passwd` using the following command using the nano editor:

```
sudo nano /etc/postfix/sasl/sasl_passwd
```

Enter the following line into the editor (on one line), substituting your email address and the generated application password:

```
[smtp.gmail.com]:587
youremail@gmail.com:yourgeneratedapppassword
```

Save the file by typing `ctl+O`, the return key, and exit nano by typing `ctl+W`.

This file contains a "clear text" password. Next, run the following two commands. The first is to protect that file. The second will generate `/etc/postfix/sasl/sasl_passwd.db` which is used by SASL.

```
sudo chmod u=rw,go= /etc/postfix/sasl/sasl_passwd
sudo postmap /etc/postfix/sasl/sasl_passwd
```

Now let's edit the `/etc/postfix/main.cf` file and configure it to use our Gmail address as an SMTP relay and enable SASL to authenticate. Enter the following command in the terminal window:

```
sudo nano /etc/postfix/main.cf
```

Search for `relayhost =` and set it to:

```
relayhost = [smtp.gmail.com]:587
```

Add the lines in Listing 1 to the end of the `main.cf`[2] file.

Save the file by typing `ctl+O`, the return key, and exit nano by typing `ctl+W`.

Restart postfix:

```
sudo systemctl restart postfix
```

Testing our Pi can send a Text Message to our Phone

Now that we have mail installed on our Pi and have configured our Gmail account to relay SMTP messages, let's test the configuration by sending an email from our Pi to our mobile phone.

To email a text message to your mobile phone, you'll need to know the email gateway your mobile phone provider uses. The two main types of gateways are Short Message Service (SMS) and Multimedia Messaging Service (MMS). MMS is the preferable service to use.

My provider is AT&T, and their MMS gateway is `mms.att.net`. These are super simple to use; just preface the gateway address with your mobile phone number just like you would with a regular email address:

```
6081234567@mms.att.net
```

Wikipedia provides a list of SMS and MMS provider gateways[3].

2 *main.cf:* http://main.cf

3 *SMS and MMS provider gateways:* https://en.wikipedia.org/wiki/SMS_gateway

Send yourself a simple text message from the terminal window using the following command and adjust the phone number and gateway according to your provider:

```
echo "This is a test." | mail -s \
"Test Message" 6081234567@mms.att.net
```

The command echoes the message, "This is a test." and pipes it to the `mail` utility that sets the subject to "Test Message" and sends it to your phone number using your provider's MMS gateway.

After a couple of seconds, I received the text message on my phone.

Great! Now we can send and receive text messages.

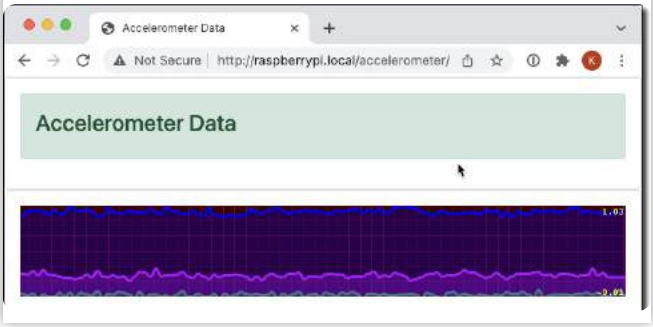
We Need a State Machine

You would think that now we could just set our Pi on top of our clothes dryer and write a simple program with an infinite loop that looks for accelerometer motion and sends us a text message when we detect motion. Right? Unfortunately, it is a bit more complicated and requires us to guard against false positives.

By false positives, I'm thinking of two things: first, the cat jumping on top of the dryer when the dryer isn't actually running, and second, the dryer doesn't always run continuously during the drying cycle; sometimes, it stops for a couple of seconds, then continues.

Let's look at the accelerometer data with my Pi on top of my dryer while the dryer is running. See Figure 6.

Figure 6. Accelerometer data Plot of Dryer Running



In my testing, I've made a few observations about what the g-force data looks like when my clothes dryer is running.

The first observation is a consistent oscillation in the Z-axis waveform while the dryer is running. In fact, when I take an 11-point moving average to smooth out the waveform data, I always have a difference of over 0.03 Gs of force between the highest and lowest points of the moving average when the dryer is running, and it is always less than 0.02 Gs of force when it is not running.

The second observation is (as previously mentioned) that the dryer doesn't always run continuously during the complete drying cycle. However, it never stops for more than 3 or 4 seconds. Therefore, I can be sure the drying cycle is complete if the dryer stops for more than 10 seconds.

These two observations give us the criteria for putting a program together to ensure we only send a text message to our mobile phone when we know that the clothes dryer has completed its drying cycle. The best way to accomplish this is to use a Finite State Machine(FSM)[4].

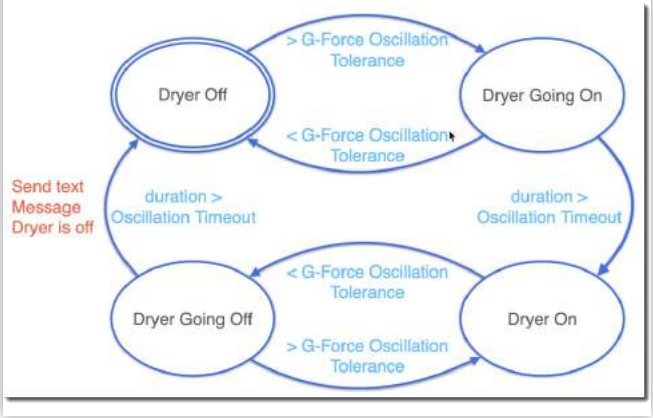
What is a Finite State Machine?

There are whole books and engineering courses devoted to State Machines, how they work, and how best to build them. In a nutshell, they define a finite set of states that can exist and that a system can be in. The system can only be in one state at a time. Each state that the system is in can have one or more events that it listens to that cause the system to transition to another state. During transitions from one state to another, various actions can be performed. The purpose of a state machine is to reduce the complexity of a system by forcing events and actions to be monitored and performed in a deterministic fashion. Typical examples of systems that use state machines are elevators and traffic lights. Please keep in mind this is an oversimplified explanation of state machines and how they can be used.

Dryer Cycle Finite State Machine

Let's start with a state machine diagram for our clothes dryer (Figure 7). The states in a state machine diagram are represented by ovals, with the starting state a double oval. The name of the state is noted within the oval (in black). The transitions between states are denoted with arrows. The name of the event that caused the transition is noted near the transition (in light blue). I've noted the action of sending a text message when the dryer is transitioning from **Dryer Going Off** to **Dryer Off**:

Figure 7. Dryer Cycle Finite State Machine



When the program starts, it will begin in the **Dryer Off** state. There is only one transition out of this state, which is detecting that the accelerometer is greater than the oscillation

4 *Finite State Machine(FSM):* https://en.wikipedia.org/wiki/Finite-state_machine

tolerance (of 0.02 Gs of force), leading to the **Dryer Going On** state. In the **Dryer Going On** state, there are two transitions. If the G-force oscillation remains greater than the tolerance for more than 10 seconds, there will be a transition to the **Dryer On** state. However, if the G-force oscillation falls below the tolerance within 10 seconds, a transition back to the **Dryer Off** state will occur. Once in the **Dryer On** state, if the G-force oscillation falls below the tolerance, a transition to **Dryer Going Off** will occur. Within the **Dryer Going Off** state, there are two transitions. If the G-force oscillation rises above the tolerance, a transition will occur back to the **Dryer On** state. However, if the G-force oscillation remains below the tolerance for more than 10 seconds, a transition to the **Dryer Off** state will occur. During this transition from **Dryer Going Off** to **Dryer Off,** we send a text message to our mobile phone letting us know the dryer cycle is complete.

Creating the Dryer Finite State Machine Code

Although writing state machine software can be complicated, I created a C++ program that uses a State Machine Library called TinyFSM[5] written by Axel Burri. The library uses generic template metaprogramming, which can be complex if you do not have a background in C++ template programming, so I'm not going to explain the library code. However, I will walk through some of the code I created and some of the subclasses I created that use the library. You will be modifying the `DryerFSM.cpp` source file to text your mobile phone a message that the dryer cycle is done.

Download the Dryer Finite State Machine Code

```
cd ~/Downloads
wget https://phpa.me/pi-notification
```

In the terminal window, within your `Downloads` folder, download the `dryerstatusnotification.zip`[6] file to your Pi using `wget`:

In the terminal window, unzip the downloaded zip file file and then do a `cd` into the the folder that was created:

```
unzip dryerstatusnotification.zip
cd dryerstatusnotification
```

The Dryer Finite State Machine (FSM) code contains the source code files shown in the table below.

File Name(s)	Description
GForceOscillation.hpp/cpp	Utility class for calculating the moving average for X, Y, and Z g-force data and calculating the g-force oscillation amplitude of the moving average by taking the difference between min and max points
tinyfsm.hpp	Tiny FSM template library used for managing and defining the state machine. See https://digint.ch/tinyfsm/doc/usage.html for how to use
fsmlist.hpp	A wrapper that allows events to be sent and processed in the FSM
DryerFSM.hpp/cpp	The States, transitions, and logic for transitioning from one state to another. These states are subclasses of DryerFSM, which are subclasses of the tinyfsm template library
dryer_status_notify.cpp	Main entry point of program. Starts the state machine then enters an Infinite loop that runs every 100 milliseconds. Reads accelerometer g-force data from the database, calculates an 11-point moving average of accelerometer data and g-force oscillation, and sends the oscillation as an event to the state machine
README.txt	Compilation instructions for compiling the dryer_status_notify state machine program
dryerstatusnotify.service	Unix Service Configuration file for installing the dryer_status_notify state machine program as a Unix Service

5 *TinyFSM: https://digint.ch/tinyfsm/*

6 *dryerstatusnotification.zip: https://phpa.me/pi-notification*

GForceOscillation.hpp/cpp

`GForceOscillation` is a utility class for calculating the moving average for smoothing out the X, Y, and Z g-force data and then calculating the g-force oscillation of each axis. Let's take a look at the header file in Listing 2.

We have two vectors. `m_mv_avg_axis_vec` for calculating the moving average of the accelerometer axis data and `m_min_max_axis_vec` for holding the moving average of the accelerometer data used for finding the minimum and maximum smoothed points for calculating the g-force oscillation. `MV_AVG_VEC_SIZE` is a class constant that holds the interval size for our moving average calculation.

Let's take a look at the `.cpp` source file in Listing 3.

Listing 3.

```
#ifndef GFORCE_OSCILLATION__CPP
#define GFORCE_OSCILLATION__CPP
#include "GForceOscillation.hpp"

// ------------------------------------------------
// Static Constant Definitions
//
const int GForceOscillation::MV_AVG_VEC_SIZE = 11;

...
#endif // GFORCE_OSCILLATION__CPP
```

First, we have an include guard for our header file, and then we set the moving average interval to 11, which will be the number of items in the vector.

Next, we have a constructor that initializes our vectors to be empty:

```
// Constructor
GForceOscillation::GForceOscillation()
{
    m_mv_avg_axis_vec.clear();
    m_min_max_axis_vec.clear();
}
```

Then we have an empty destructor:

```
// Destructor
GForceOscillation::~GForceOscillation()
{
} appendMovingAverageData(tAxis axis)
```

Listing 5.

```
bool GForceOscillation::isMovingAverageDataAvailable()
{
    bool retVal = false;

    if (m_mv_avg_axis_vec.size() == MV_AVG_VEC_SIZE)
    {
        retVal = true;
    }

    return retVal;
}
```

Listing 2.

```
#ifndef GFORCE_OSCILLATION__HPP
#define GFORCE_OSCILLATION__HPP
#include <vector>

using namespace std;

// accelerometer axis structure
typedef struct axis
{
    float X = 0.0;
    float Y = 0.0;
    float Z = 0.0;
}tAxis;

typedef vector<tAxis> tAxisVec;

// --------------------------------
// GForceOscillation (class) declaration
//
class GForceOscillation
{
public:
    GForceOscillation();
    ~GForceOscillation();

    void appendMovingAverageData(tAxis axis);

    bool isMovingAverageDataAvailable();
    tAxis getMovingAverage();

    void appendMinMaxData(tAxis axis);

    bool isMinMaxDataAvailable();
    tAxis getMinMaxDiff();
protected:
    tAxisVec m_mv_avg_axis_vec;
    tAxisVec m_min_max_axis_vec;
    static const int MV_AVG_VEC_SIZE;
};

#endif // GFORCE_OSCILLATION__HPP
```

which appends the accelerometer data onto the end of the `m_mv_avg_axis_vec` vector removing the first entry if the vector is greater than `MV_AVG_VEC_SIZE`. See Listing 4.

Listing 4.

```
void GForceOscillation::appendMovingAverageData(tAxis axis)
{
    m_mv_avg_axis_vec.push_back(axis);

    if (m_mv_avg_axis_vec.size() > MV_AVG_VEC_SIZE)
    {
        m_mv_avg_axis_vec.erase(m_mv_avg_axis_vec.begin());
    }
}
```

Listing 7.

```
1. void GForceOscillation::appendMinMaxData(tAxis axis)
2. {
3.     m_min_max_axis_vec.push_back(axis);
4.
5.     if (m_min_max_axis_vec.size() > MV_AVG_VEC_SIZE)
6.     {
7.         m_min_max_axis_vec.erase(m_min_max_axis_vec.begin());
8.     }
9. }
```

Listing 9.

```
1. tAxis GForceOscillation::getMinMaxDiff()
2. {
3.     tAxis retVal;
4.     tAxis min;
5.     tAxis max;
6.
7.     vector<tAxis>::const_iterator iter;
8.
9.     for (iter = m_min_max_axis_vec.begin();
               iter != m_min_max_axis_vec.end(); iter++)
10.    {
11.        if (iter == m_min_max_axis_vec.begin()) {
12.            min.X = max.X = iter->X;
13.            min.Y = max.Y = iter->Y;
14.            min.Z = max.Z = iter->Z;
15.        } // END IF 1st iteration
16.        else {
17.            if (iter->X < min.X) {
18.                min.X = iter->X;
19.            } else if (iter->X > max.X) {
20.                max.X = iter->X;
21.            }
22.
23.            if (iter->Y < min.Y)
24.            {
25.                min.Y = iter->Y;
26.            } else if (iter->Y > max.Y) {
27.                max.Y = iter->Y;
28.            }
29.
30.            if (iter->Z < min.Z) {
31.                min.Z = iter->Z;
32.            } else if (iter->Z > max.Z) {
33.                max.Z = iter->Z;
34.            }
35.        } // END ELSE all other iterations except the 1st
36.    } // END FOR EACH min_max_axis_vec item
37.
38.    retVal.X = max.X - min.X;
39.    retVal.Y = max.Y - min.Y;
40.    retVal.Z = max.Z - min.Z;
41.
42.    return retVal;
43. }
44.
45. #endif // GFORCE_OSCILLATION__CPP
```

`isMovingAverageDataAvailable()` (Listing 5) is used to make sure we have enough points for calculating the moving average:

Listing 6.

```
1. tAxis GForceOscillation::getMovingAverage()
2. {
3.     tAxis total;
4.
5.     tAxis retVal;
6.
7.     vector<tAxis>::const_iterator iter;
8.
9.     for (iter = m_mv_avg_axis_vec.begin();
               iter != m_mv_avg_axis_vec.end(); iter++)
10.    {
11.        total.X += iter->X;
12.        total.Y += iter->Y;
13.        total.Z += iter->Z;
14.    }
15.
16.    if (total.X != 0.0 && total.Y != 0.0 && total.Z != 0.0)
17.    {
18.        retVal.X = total.X / m_mv_avg_axis_vec.size();
19.        retVal.Y = total.Y / m_mv_avg_axis_vec.size();
20.        retVal.Z = total.Z / m_mv_avg_axis_vec.size();
21.    }
22.
23.    return retVal;
24. }
```

`getMovingAverage()` (Listing 6) calculates and returns the moving average for each axis.

`appendMinMaxData(tAxis axis)` appends the moving average of the accelerometer data onto the end of the `m_min_max_axis_vec` vector removing the first entry if the vector is greater than `MV_AVG_VEC_SIZE`. This vector is used for getting the minimum and maximum points for calculating a g-force oscillation value, so we know if the dryer is running or not. See Listing 7.

`isMovingAverageDataAvailable()` (Listing 8) is used to make sure we have enough points for getting the minimum and maximum points for calculating the g-force oscillation.

`getMinMaxDiff()` (Listing 9) is the method that calculates the minimum and maximum points for the moving averages

Listing 8.

```
1. bool GForceOscillation::isMinMaxDataAvailable()
2. {
3.     bool retVal = false;
4.
5.     if (m_min_max_axis_vec.size() == MV_AVG_VEC_SIZE)
6.     {
7.         retVal = true;
8.     }
9.
10.    return retVal;
11. }
```

and calculates and returns the difference between the minimums and maximums for each axis. These are the g-force oscillation values for each axis.

tinyfsm.h

`tinyfsm`[7] is the template library written by Axel Burri that provides the framework for creating finite state machine code. `tinyfsm::Fsm` is subclassed by `DryerFSM` to create the dryer FSM.

fsmlist.hpp

Our dryer state machine transitions between states based on the g-force oscillation. This g-force oscillation is an event that needs to be dispatched to the state machine every 100 milliseconds. For events to be made available for dispatch to the state machine, the method `send_event()` must be defined in the `fsmlist.hpp` header file shown in Listing 10. This code will make more sense later when you see it called in the main loop of `dryer_status_notify.cpp`.

Method	Purpose
react(GForceOscillationEvent const &)	The `react()` method will pass g-force oscillation events to the active state every 100 milliseconds
entry()	The `entry()` method is called when a transition occurs into this state. This method is used to execute actions to be performed upon entry into this state
exit()	The `exit()` method is called when a transition occurs out of this state. This method is used to execute actions to be performed upon exiting this state

Listing 10.

```
#ifndef FSMLIST__HPP
#define FSMLIST__HPP

#include "tinyfsm.hpp"

#include "DryerFSM.hpp"

typedef tinyfsm::FsmList<DryerFSM> fsm_list;

/* wrapper to fsm_list::dispatch() */
template<typename E>
void send_event(E const & event)
{
    fsm_list::template dispatch<E>(event);
}

#endif //FSMLIST__HPP
```

DryerFSM.hpp/cpp

`DryerFSM` is a subclass of `tinyfsm::Fsm`. `DryerFSM` will be the subclass for each state in our state machine. `DryerFSM` must define the following three methods:

Now, let's take a look at the header file (Listing 11).

In addition to the `react()`, `entry()`, and `exit()` methods, note that our g-force oscillation event is a subclass of `tinyfsm::Event`. Also, notice I am using the `<chrono>` library to keep track of the number of seconds the -g-force oscillation remains over or under a tolerance value indicating the dryer is on or off. `GFORCE_OSCILLATION_TOLLERANCE`, `startGForceOscillationTolleranceTime`, and `GFORCE_OSCILLATION_TIMEOUT` are used to keep track of this.

Let's take a look at the `.cpp` source file shown in Listing 12.

Listing 11.

```
#ifndef DRYERFSM__HPP
#define DRYERFSM__HPP

#include "tinyfsm.hpp"
#include <chrono>

// ------------------------------------
// Event declarations
//

struct GForceOscillationEvent : tinyfsm::Event
{
    float gforceOscillation;
};

// ------------------------------------
// DryerFSM (FSM base class) declaration
//
class DryerFSM : public tinyfsm::Fsm<DryerFSM>
{
public:
    // Default reaction for unhandled events
    void react(tinyfsm::Event const &) {};

    virtual void react(GForceOscillationEvent   const &);
    virtual void entry(void) {}; // entry actions in some states
    virtual void exit(void) {}; // Maybe exit actions in some states

protected:
    static const float GFORCE_OSCILLATION_TOLLERANCE;
    static const int GFORCE_OSCILLATION_TIMEOUT; // In seconds
    static std::chrono::steady_clock::time_point \
                startGForceOscillationTolleranceTime;
    // Use: std::chrono::steady_clock::now(); // to set current time
};

#endif // DRYERFSM__HPP
```

7 *tinyfsm:* https://digint.ch/tinyfsm/

Listing 12.

```
#include <iostream>
#include <stdlib.h>
#include "tinyfsm.hpp"
#include "DryerFSM.hpp"
#include "fsmlist.hpp"

#define DRYER_STATUS_SEND_TEXT_MSG_ENABLED true

using namespace std;
```

First we need to include `<iostream>` and `<stdlib>` for access to I/O streaming and the C++ library. Then we include the state machine library, or dryer FSM header, and `fsmlist.hpp`, which allows us to dispatch g-force oscillation events to our state machine.

Then we have a definition that allows you to enable or disable sending a text message to your mobile phone when the dryer cycle is complete.

Finally, we'll indicate we are using the `std::` namespace for C++ standard library functions, so we don't have to preface all those calls with `std::`.

Next, we create four declarations for the states of our dryer state machine, which should look familiar based on our description of the states back in Figure 7:

```
...
class DryerOff;
class DryerGoingOn;
class DryerOn;
class DryerGoingOff;
```

Then we have class constant definitions for our g-force tolerance and the g-force timeout set to the values we discussed above:

```
...
const float DryerFSM::GFORCE_OSCILLATION_TOLLERANCE = 0.02;

// In Seconds
const int DryerFSM::GFORCE_OSCILLATION_TIMEOUT = 10;
```

Now we define our states with transitions which will be subclasses of `DryerFSM`. Let's take a look at `DryerOff` in Listing 13. All of our states will implement the `entry()` method, which I am using to indicate which state we have entered by streaming it to standard output. The `react()` method is what gets called every 100 milliseconds and contains the g-force oscillation as an event. If the g-force oscillation is greater than the tolerance, we transition to the `DryerGoingOn` state.

Listing 13.

```
// ----------------
// State: DryerOff
//
class DryerOff : public DryerFSM
{
  void entry() override
  {
    cout << "Dryer: Off" << endl;
  }

  void react(GForceOscillationEvent const & e) override
  {
    if (e.gForceOscillation >
        GFORCE_OSCILLATION_TOLLERANCE) {
      transit<DryerGoingOn>();
    }
  }
};
```

Let's take a look at `DryerGoingOn` in Listing 14. When we enter the `DryerGoingOn` state, we set `startGForceOscillationTolleranceTime` to the current time. Every time the `react()` method is called, we note how much time has elapsed since we set `startGForceOscillationTolleranceTime`. Then we check to see if the g-force oscillation has dropped below the tolerance for the dryer being on and transition back to `DryerOff` if so. Otherwise, we check to see if the duration we have been in this state is greater than the oscillation timeout (10 seconds). If so, we transition to `DryerOn`.

Listing 14.

```
class DryerGoingOn : public DryerFSM
{
  void entry() override // Start timer
  {
    cout << "Dryer: Going On" << endl;
    startGForceOscillationTolleranceTime =
        chrono::steady_clock::now();
  }

  void react(GForceOscillationEvent const & e) override
  {
    chrono::steady_clock::time_point nowTime =
      chrono::steady_clock::now();
    chrono::steady_clock::duration timeDiff =
      nowTime - startGForceOscillationTolleranceTime;

    int duration =
      chrono::duration_cast<chrono::seconds>(timeDiff).count();

    if (e.gForceOscillation < GFORCE_OSCILLATION_TOLLERANCE)
    {
      transit<DryerOff>();
    } else if (duration > GFORCE_OSCILLATION_TIMEOUT) {
      transit<DryerOn>();
    }
  }
};
```

Let's take a look at `DryerOn` in Listing 15, where after we enter the `DryerOn` state, we are looking for the g-force oscillation to drop below the tolerance indicating the dryer is off. If so, we transition to the `DryerGoingOff` state.

Listing 16 looks at `DryerGoingOff`. When we enter the `DryerGoingOff` state, we set `startGForceOscillationTolleranceTime` to the current time. Every time the `react()` method is called, we note how much time has elapsed since we set `startGForceOscillationTolleranceTime`. Then we check to see if the g-force oscillation has increased above the tolerance for the dryer being on and, if so, transition back to `DryerOn`. Otherwise, we check to see if the duration we have been in this state is greater than the oscillation timeout (10 seconds). If so, we make a system call using the `mail` program to send a text message to our mobile phone and then transition back to `DryerOff`.

This single line of code is where you will make your modifications for your SMS/MMS gateway and mobile phone number:

```
system("echo \"Dryer is Off.\" |
mail -s \"Dryer Status\" 6081234567@mms.att.net");
```

Finally, we need to define a default implementation of the `react()` method, set

Listing 15.

```
// ----------------------------------------------------
// State: DryerOn
//
class DryerOn : public DryerFSM
{
    void entry() override
    {
        cout << "Dryer: On" << endl;
    }

    void react(GForceOscillationEvent const & e) override
    {
       if (e.gForceOscillation < GFORCE_OSCILLATION_TOLLERANCE)
        {
            transit<DryerGoingOff>();
        }
    }
};
```

Listing 16.

```
// ----------------------------------------------------------------------
// State: DryerGoingOff
//
class DryerGoingOff : public DryerFSM
{
    void entry() override
    {
        cout << "Dryer: Going Off" << endl;

        startGForceOscillationTolleranceTime = chrono::steady_clock::now();
    }

    void react(GForceOscillationEvent const & e) override
    {
        chrono::steady_clock::time_point nowTime = chrono::steady_clock::now();

        chrono::steady_clock::duration timeDiff = nowTime - startGForceOscillationTolleranceTime;

        int duration = chrono::duration_cast<chrono::seconds>(timeDiff).count();

        if (e.gForceOscillation > GFORCE_OSCILLATION_TOLLERANCE)
        {
            transit<DryerOn>();
        }
        else if (duration > GFORCE_OSCILLATION_TIMEOUT)
        {
            #ifdef DRYER_STATUS_SEND_TEXT_MSG_ENABLED
                        cout << "\tSend notification: Dryer is Off" << endl;
            system("echo \"Dryer is Off.\" | mail -s \"Dryer Status\" 6081234567@mms.att.net");
            #endif // DRYER_STATUS_SEND_TEXT_MSG_ENABLED

            transit<DryerOff>();
        }
    }
};
```

Listing 18.

```
// Constants
#define HOST "localhost"
#define USER "accelerometer"
#define PASSWD "accelerometer"
#define DB "AccelerometerData"

MYSQL *connection, mysql;
MYSQL_RES *result;
MYSQL_ROW row;
int query_state;

using namespace std;
```

Listing 17.

```
// ------------------------------------------------------------------------
// Base state: Default implementations
//

void DryerFSM::react(GForceOscillationEvent const &)
{
    cout << "Call event ignored" << endl;
}

chrono::steady_clock::time_point DryerFSM::startGForceOscillationTolleranceTime
        = std::chrono::steady_clock::now();

// ------------------------------------------------------------------------
// Initial State Definition
//
FSM_INITIAL_STATE(DryerFSM, DryerOff);
```

Listing 19.

```
int main()
{
    cout << "Dryer FSM Status" << endl;
    cout << endl;

    GForceOscillation g_force_oscillation;

    fsm_list::start();

    GForceOscillationEvent g_force_oscillation_event;

    mysql_init(&mysql);

    // the three zeros are: Which port to connect to, which socket to connect to
    // and what client flags to use.
    // Unless you're changing the defaults you only need to put 0 here
    connection = mysql_real_connect(&mysql,HOST,USER,PASSWD,DB,0,0,0);

    // Report error if failed to connect to database
    if (connection == NULL)
    {
        cout << mysql_error(&mysql) << endl;
        return 1;
    }
        ...
}
```

`startGForceOscillationTolleranceTime` to the current time when the state machine is instantiated, and set the initial state to `DryerOff`. See Listing 17.

dryer_status_notify.cpp

`dryer_status_notify.cpp` is the main routine that starts the state machine and enters an infinite loop that runs every 100 milliseconds sending g-force oscillation data as an event to the state machine. Let's walk through the code:

```
#include <stdlib.h>
#include <iostream>
#include <memory>
#include <unistd.h>
#include <stdexcept>
#include <string>
#include "/usr/include/mysql/mysql.h"
```

First, we have the required includes for using the standard C++ library, memory, I/O streaming, error handling, and connecting the MySQL.

Next we have includes for calculating g-force oscillation and our state machine:

```
#include "GForceOscillation.hpp"
#include "fsmlist.hpp"
```

In Listing 18, we have the credentials and references for querying our accelerometer g-force data and we are using the `std::` namespace.

Now let's take a look at the `main()` routine in Listing 19.

First, we'll stream a message to standard out, instantiate our `GForceOscillation` calculation object, `GForceOscillationEvent`, and connect to the `AccelerometerData` database.

Next, we'll enter our infinite loop (Listing 20), which will run every 100 milliseconds.

In every iteration of our loop, we will query the database for the latest acceleration g-force data (Listing 21).

Then we will append the g-force data for all axes to our GForceOscillation object `g_force_oscillation` so we can calculate moving averages as shown in Listing 22.

Once we have enough points available to calculate moving averages, we will get the latest moving average point and append it to the MinMax vector in the `g_force_oscillation` object that will hold the moving averages. We can then use the vector to find the minimum and

Listing 21.

```
...
while(1) {
    string newest_entry_query = "SELECT axis_x, axis_y, axis_z
       FROM accelerometer_data ORDER BY created DESC LIMIT 1";

    // Send newest entry query to database
    query_state = mysql_query(connection, newest_entry_query.c_str());

    if(query_state != 0)
    {
        cout << mysql_error(connection) << endl;
        return 1;
    }

    // Get result of query (newest axis data)
    result = mysql_store_result(connection);

    row = mysql_fetch_row(result);

    tAxis axis;
    axis.X = stof(row[0]);
    axis.Y = stof(row[1]);
    axis.Z = stof(row[2]);

    mysql_free_result(result); // Avoid memory leak
        ...
} // End forever loop
...
```

Listing 20.

```
int main()
{
        ...
        // Forever Loop
    while(1)
    {
                ...
                usleep(100000); // 100 milliseconds
    } // End forever loop

    cout << "Done." << endl;
    return EXIT_SUCCESS;
}
```

Listing 22.

```
...
while(1)
{
        ...
    // Free result or a memory leak will occur
    mysql_free_result(result);


        g_force_oscillation.appendMovingAverageData(axis);
        ...
} // End forever loop
...
```

maximum points for getting our oscillation value which we send as an event to the dryer state machine by calling `send_event()`. See Listing 23.

Notice that we are only sending the g-force oscillation for the Z-axis since I assume you will place your accelerometer with the Z-axis pointing up on the dryer. Also, notice there is some commented-out code for debugging if you need it.

Edit the phone number in DryerGoingOff within DryerFSM.cpp

Use the `nano` editor to open up the `DryerFSM.cpp` file in the terminal window. Locate the `react()` method within the `DryerGoingOff` class and modify the mobile phone number and the SMS/MMS gateway to match your mobile phone provider's:

```
system("echo \"Dryer is Off.\" |
mail -s \"Dryer Status\" phone@mms.att.net");
```

Save the file by typing `ctl+O`, the return key, and exit nano by typing `ctl+W`.

Compile and run the State Machine

If you take a look at the `README.txt` file, you will see the command needed to compile the dryer state machine code into an executable file called `dryer_status_notify`:

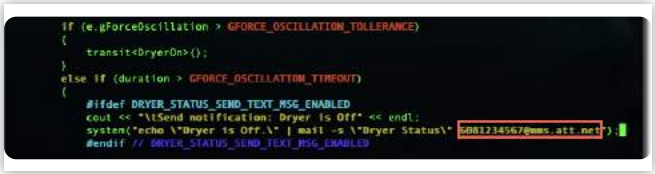
```
g++ -std=c++11 -o dryer_status_notify dryer_status_notify.cpp \
GForceOscillation.cpp DryerFSM.cpp -lmysqlclient -lmysqlcppconn
```

Listing 23.

```
while(1) {
    ...
  g_force_oscillation.appendMovingAverageData(axis);
  if (g_force_oscillation.isMovingAverageDataAvailable()) {
    g_force_oscillation.appendMinMaxData(
       g_force_oscillation.getMovingAverage()
      );
   if (g_force_oscillation.isMinMaxDataAvailable()) {
            tAxis min_max_diff =
                    g_force_oscillation.getMinMaxDiff();

            // We are only concerned with the Z axis
            // difference for driving our state machine
            g_force_oscillation_event.gForceOscillation =
                    min_max_diff.Z;

            send_event(g_force_oscillation_event);

            /*
            cout << "Min/Max Diff: X: " << min_max_diff.X
                    << ", Y: " << min_max_diff.Y
                    << ", Z: " << min_max_diff.Z
                    << endl;
            */
        }
    }
    usleep(100000); // 100 milliseconds} // End forever loop
} // End forever loop
...
```

Copy and paste this line into the terminal window and execute the compilation. After about 20 seconds, the bash prompt should return without an error. If you do a long listing (`ls -al`), you should see the executable program `dryer_status_notify`:

Okay, so the moment of truth has finally arrived for us to test if we can really send a text message that the dryer cycle is done from our Raspberry Pi. Start the dryer state machine program by typing the following in the terminal window:




```
./dryer_status_notify
```

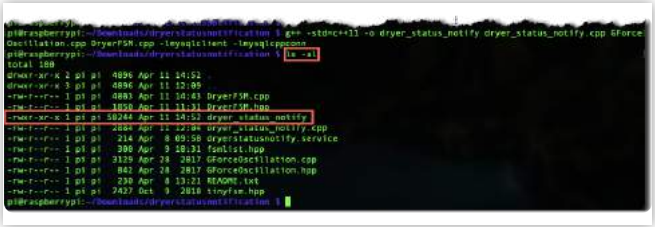
You should see the `Dryer FSM Status` opening program message and `Dryer: Off` message displayed in the terminal window, indicating we are in the *DryerOff* state:

To properly visualize this, bring up a browser tab on your local computer and navigate to `http://raspberrypi.local/accelerometer`. Shake the accelerometer, but not for more than a couple of seconds:

In the terminal window, you will notice the state machine transitions from *Off* to *Going On* back to *Off*:

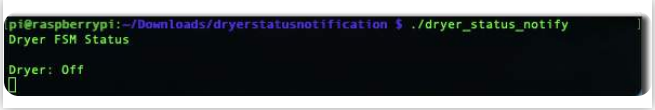
Now shake the accelerometer for over 10 seconds until you see the state machine transition to the *Dryer On* state:

In the terminal window, you will notice the state machine transitions from *Off* to *Going On* to *On*. At this point, if you stop shaking the accelerometer, the state machine will transition to the *Going Off* state. If you start shaking the



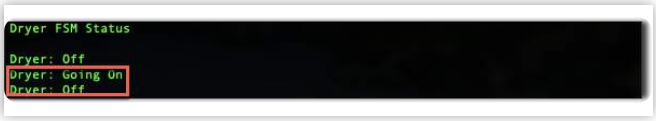
accelerometer again before 10 seconds have elapsed, the state machine will transition back to the *On* state. However, if you stop shaking the accelerometer, and don't shake it for more than 10 seconds; the state machine will send an email that text's your mobile phone and transition back to the *Off* state:

After a couple of seconds, you should get the following text message on your mobile phone:



Install dryer_status_notify as a Unix Service

Let's create a Unix service that loads the`dryer_status_notify` program when the Raspberry Pi boots and unloads it when the Pi shuts down.

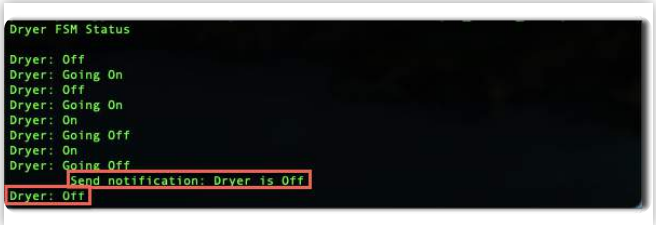


> 💡 *Please see part 3 of this series under the Creating and Installing a Unix Service section for a more detailed understanding of Unix services.*

The dryerstatusnotify.service Configuration File

I have already created the `dryerstatusnotify.service` Unix service Unit configuration file for you in Listing 24.

Note that `StandardOutput` is set to `null` to disable the state transition output to the terminal window while the service is running.



Installing and Enabling our Service

For systemd to find and run our service, we need to copy our executable program to the `/root` folder and copy our unit

service configuration file to the `/lib/systemd/system` folder. In the terminal window, type the following commands:

```
sudo cp dryer_status_notify /root/
sudo cp dryerstatusnotify.service /lib/systemd/system/
```

To enable our service using `systemctl`, type the following in the terminal window:

```
sudo systemctl enable dryerstatusnotify.service
```

Doing so will create a symbolic link that systemd will use to start and stop our dryer status notification service. You should see the following displayed in the terminal window:

You can now start the service using `systemctl` by typing the following in the terminal window:

```
sudo systemctl start dryerstatusnotify.service
```

If you pick up your accelerometer and shake it for more than 10 seconds, you should get a text message on your mobile phone letting you know the dryer cycle is complete.

Conclusion

Now you can put your Raspberry Pi on top of your clothes dryer and receive a text notification on your mobile phone every time your dryer cycle is complete!

It has been an absolute privilege to share my knowledge of creating an Internet of Things (IoT) device using the Raspberry Pi with you. I hope this article series opens your mind to more ideas and possible applications you can create using the Raspberry Pi.

This concludes our series; however, know that there are

```
pi@raspberrypi:~/Downloads/dryerstatusnotification $ sudo systemctl enable dryerstatusnotify.service
Created symlink /etc/systemd/system/dryerstatusnotify.service → /lib/systemd/system/dryerstatusnotify.service.
Created symlink /etc/systemd/system/multi-user.target.wants/dryerstatusnotify.service → /lib/systemd/system/dryerstatusnotify.service.
pi@raspberrypi:~/Downloads/dryerstatusnotification $
```

many more sensors and devices you can connect to the Pi, and I hope this series has inspired you to dust off your little IoT computer now and again now that you know how to *Hack your Home with a Raspberry Pi*!

Listing 24.

```
1. [Unit]
2. Description=Dryer Status Notification Service
3. After=multi-user.target
4.
5. [Service]
6. ExecStart=/root/dryer_status_notify
7. StandardOutput=null
8.
9. [Install]
10. WantedBy=multi-user.target
11. Alias=dryerstatusnotify.service
```

Ken Marks has been working in his dream job as a Programming Instructor at Madison College in Madison, Wisconsin, teaching PHP web development using MySQL since 2012. Prior to teaching, Ken worked as a software engineer for more than 20 years, mainly developing medical device software. Ken is actively involved in the PHP community, speaking and teaching at conferences. @FlibertiGiblets

Continuous Code

Chris Tankersley

One of the greatest accomplishments of the web was the ability to quickly deliver software to users. While you can still technically go to the store and purchase software on a solidified blob of plastic that can be read by lasers, the world has moved on to digital delivery of almost everything. Music, movies, software, and even television is delivered over the internet.

Until the internet had invaded every part of our lives, we had to distribute software in a very static form, and users had to live with whatever bugs existed. If you were lucky, you might get a patch release, but many users had to wait until a completely brand new version of whatever, be it a business application or software, came out.

In the wonderful year 2022, software is constantly patched. Most of the time, if you purchase software in physical form, it will immediately update itself to the newest version the first time it gets a chance. Purchase a video game at the store? Enjoy that 50-gigabyte first-day patch that needs to download.

On top of that, much of the software that we use are Software-as-a-Service products. These services can update multiple times on the same day. Even way back in 2009, Flickr deployed up to 10 times per day[1]. My day job has us pushing updates every few days, with a few days having multiple deploys. Continuously delivering code is now the norm.

For many developers, this is a scary prospect. So many developers have been burned by a deployment that took out production that the idea of having code automatically go to users without someone pushing a button seems impossible. If you are afraid to deploy on a Friday, how in the world will you feel comfortable enough to deploy all the time?

There is an extreme lack of confidence in much of the code that we write. The good news is that you can build up that confidence over time. Let's look at what it takes to build up to where code can go out to your users without you worrying.

Automated Testing

The first step toward Continuous Delivery is to automate the testing phase of development. Many older pieces of software used a design process called Big Design Up Front. With Big Design Up Front or BDUF, the core idea was that an early design phase would smooth out issues or bugs that could crop up later. BDUF may have helped remove logic bugs like those around user workflows or how some processes might interact. But it did nothing for the actual bugs we as developers write into code.

Some people, like IBM software engineer Grady Booch[2], proposed a much more iterative design process known as the "Booch Method." His method was outlined in his book "Object-Oriented Analysis and Design with Applications." The software would be iterated on in incremental stages and worked toward completeness.

Iterative development allowed tests to happen earlier in the software design lifecycle, and bugs could be caught quicker. Companies like Rational Software, where Booch was employed, helped design software to work with this iterative process. For many developers, manual testing was still the norm even though the idea of iterative development was an early 1990s idea.

As Extreme Programming gained popularity in the late nineties, so did the idea of automated testing. Testing was a core ideal of Extreme Programming, and automated testing tools helped speed up the feedback loop for bugs and regressions. Both unit tests, or tests of individual components in isolation, as well as acceptance tests, or tests against usage requirements between multiple components, saw new tools appear to help with the automation process.

> *Why did automated testing take so long to come around? It was very expensive to write software in the early days, and much of the software that was written only worked for the system it was designed for. While languages like C were designed to be multi-platform, it still took time to make things work for the multitude of platforms available. On top of that, it was not only three players like today (Windows, macOS, and Linux); you had a huge number of hardware and software combinations to make work.*

Extreme Programming also fostered the test-first philosophy, or what we normally call Test-Driven Development. In Test-Driven Development, TDD, the developer writes the test before implementing any business logic. The developer will verify the tests fail, implement code, then see if the tests pass. Once the tests pass, the developer can move on, or continue to refactor.

As a quick run-through of Test-Driven Design in PHP, consider a class that fetches the current weather for a location. How would we do Test-Driven Design to write this code?

First, we need something that helps us write tests. I suggest starting off with PHPUnit. We can install that as a development dependency with Composer:

```
$ composer require --dev phpunit/phpunit ^9
```

1 *10 times per day: https://phpa.me/slideshare-10-deploys-day*

2 *Grady Booch: https://phpa.me/ibm-gbooch*

Next, we need to decide what we want to test. Since we need some way to get the weather for a location, we need to work with an API. OpenWeatherMap[3] has an easy-to-use API where we can get the information we need. We should have an `OpenWeatherClient` class that encapsulates all our API calls.

Since we want to get the weather for a location, we will want a `getWeather()` method that takes a zip code as the string parameter `$zipcode`. It will return the current temperature and weather description as a string response.

Now because of how PHP works, let's sketch out the class we want to test. In some languages, you would start to test right away, but PHP will complain if the class and method do not exist. For the sake of our example, we will put it in `src/OpenWeatherClient.php`

```
class OpenWeatherClient
{
    public function getWeather(string $zipcode): string
    {
    }
}
```

Notice that we did not put any logic in the class. We just need the class and method declared. We are not writing any business logic just yet. Since we have our class and method, we need a test. Let's make a new file (Listing 1) in `src/OpenWeatherClientTest.php` with a class that extends `PHPUnit\Framework\TestCase`.

We can now run our tests against our `src/` directory and test. See Listing 2.

We get our first error! Since this is not a full-on tutorial on Test-Driven Development, we'll leave it here, but your next steps would be to resolve that error. Our `getWeather()` method needs to return a string that matches our regex; keep running your tests until everything passes.

A lot can go into Test-Driven Development, so I highly suggest reading over the PHPUnit documentation. "The Grumpy Programmer's Guide to Testing PHP Applications"[4] is available through the fine folks that also put together this magazine. There are also plenty of resources available on PHP: The Right Way[5], including other testing methodologies.

Listing 1.

```
1. require_once 'OpenWeatherClient.php';
2.
3. use PHPUnit\Framework\TestCase as TestCase;
4.
5. class OpenWeatherClientTest extends TestCase
6. {
7.     public function testGetWeather()
8.     {
9.         $client = new OpenWeatherClient();
10.        $weather = $client->getWeather('20001');
11.        $match = '/It is \\d+F and [a-zA-Z0-9_ ].*/';
12.        $this->assertMatchesRegularExpression($match, $weather);
13.    }
14. }
```

The main idea here is to start moving your tests into something that can be run automatically and get away from manual testing. Manually testing your website whenever you deploy can take an incredible amount of time. To truly test your website, you would need to test every feature of your website every single time code gets pushed. Even with a dedicated team, it will take a monumental amount of time; time that is better spent writing code.

Making your tests automated also lets anyone, at any point, test all the code that is covered. For example, in my day job, we use integration testing to test all the features a user can do on the website. I can work on my feature request, then with a single command test that 76 workflows with 442 steps work correctly. I can verify my code works, as well as all the other code.

Listing 2.

```
$ vendor/bin/phpunit src/
PHPUnit 9.5.20 #StandWithUkraine

E                                                                   1 / 1 (100%)

Time: 00:00.013, Memory: 4.00 MB

There was 1 error:

1) OpenWeatherClientTest::testGetWeather
TypeError: OpenWeatherClient::getWeather(): Return value must be of type string, none returned

/Users/ctankersley/Projects/tmp/src/OpenWeatherClient.php:7
/Users/ctankersley/Projects/tmp/src/OpenWeatherClientTest.php:14

ERRORS!
Tests: 1, Assertions: 0, Errors: 1.
```

3 *OpenWeatherMap:* https://openweathermap.org

4 *"The Grumpy Programmer's Guide to Testing PHP Applications":* https://phpa.me/grumpy-testing-book

5 *PHP: The Right Way:* https://phptherightway.com/#test_driven_development

So step one of getting more confident in your code is automating your tests. Know that you and anyone on your team can verify, at any time, that the code is working. In this initial phase, you may want to take a two-phase approach. Run the test suite against your local code and a copy of the website. If that looks good, deploy your code. If that all works, run as many integration tests as you can against the newly deployed code.

As you build up your suite of tests, your confidence in the code will grow. Unit tests help validate small chunks of code work as expected, and integration tests will help validate that interdependent systems function as they should. Do not forget that you can also automate your integration tests, linters, and code sniffers.

Continuous Integration

Now that we have our code in the beginning stages of being able to test, the next jump is to have the tests run automatically without any intervention. If we go back to "Object-Oriented Analysis and Design with Applications," one of the core ideas for writing maintainable code was early and frequent integration.

Right now, our tests are automated, but it still requires someone to invoke the `phpunit` command. While you will still want to run all your tests and tools locally, we also want to run those commands and tools automatically when we publish pull requests or merge code together.

We run these tools constantly as new code is pushed. This is where the "Continuous" in "Continuous Integration" comes in. We constantly test that new code integrates with the old code, doing so in an automated manner, allowing us to test and validate new code early and often.

A big win is just getting automated testing running and reporting on pull requests. The idea is when we push code up into the public eye, say as a pull request, another system is alerted and kicks off our testing routines. Over the last few years, one of the easiest ways to set this up has been to use GitHub Actions if your project is hosted on GitHub.

Github Actions is a newer service provided by GitHub that should be available to everyone at this point. At its heart, GitHub Actions is just a task runner that uses a YAML file to tell it what commands to run. There is a marketplace with many free workflows that can help cut down on boilerplate, and you can write your own actions in JavaScript.

GitHub Actions can look a bit intimidating, but most of that can be attributed to a lot of boilerplate for many tasks that you may want to do. For many common tasks, like running Composer and PHPUnit, someone has already designed an existing action that you can use.

As a quick example, let's say all you want to do is run PHPUnit. There is already an action that sets up and executes PHPUnit for us[6]. We can use that along with an action that automatically checks out our code. A third action exists for running Composer, so we can put all that together to form a fairly short workflow.

GitHub Actions looks for a series of YAML files inside a special folder. Inside your repository, you will want to make the path of `.github/workflows` from the root of your repository. Inside here, we will add a new file named `ci.yml`. You can name it whatever you want, but try and name it somewhat centered around what it does.

Listing 3.

```
1. name: CI
2.
3. on: [push]
4.
5. jobs:
6.   build-test:
7.     runs-on: ubuntu-latest
8.
9.     steps:
10.    - uses: actions/checkout@v2
11.    - uses: php-actions/composer@v5
12.    - uses: php-actions/phpunit@v3
```

GitHub Action workflow files generally have three sections: a `name` section that provides a nice readable name; an `on` section that tells GitHub Actions when to run; a `jobs` section to tell GitHub Actions what to do during a run.

In our case, we will run our action any time code is pushed to the repository. You could have this run only when branches are pushed or against specific branches. For our case, we want our CI workflow to run anytime someone pushes any code.

The `job` section outlines three actions for us to take. First, we will run `actions/checkout@v2` to check out our code. Then we will run `php-actions/composer@v5` to automatically install all of our dependencies through Composer. Finally, we will run PHPUnit via `php-actions/phpunit@v3`. The outcome of this will be stored in the `Actions` tab in our repository, and linked in the Pull Request if we make one.

Again, this guide is meant more as a general signposting guide than getting deep into any one specific step, but here we are using pre-packaged actions that the community has developed. We could drop all the way down to a list of bash commands if we really wanted. The main idea is that we want our tests to run automatically anytime anyone pushes code.

With something like this, we now help satisfy one of the major tenants of Extreme Programming as the rules for writing maintainable code. We are running our tests without any intervention every time someone pushes code, not just when a single person decides to test things. As your testing gets more involved, you can add new steps to the workflow.

For example, you may want to run PHPStan[7] only on branches but always run unit tests when any code is pushed. You can even go further and write a workflow that tries to

6 *executes PHPUnit for us:* https://github.com/marketplace/actions/phpunit-php-actions

7 *PHPStan:* https://phpstan.org

merge all pull requests together and run all tests to make sure new, unmerged code does not have any conflicts.

Do not worry if you are not hosting your code on GitHub. Most hosting platforms may have their own Continuous Integration systems, sometimes called Pipelines or something similar. You may want to look into a third-party service like CircleCI[8] if your platform does not have something built-in.

No matter what continuous integration workflow you go with, you now have the confidence that testing itself brings as well as the additional automatic feedback of the workflows.

Continuous Delivery

To be fully honest, most people will be happy with stopping at Continuous Integration. The simple fact that at any time you know your code is up-to-snuff and working is enough for most people, and many people like to schedule releases. Pushing that deploy button is extremely satisfying.

The next step after testing your code automatically and frequently is to push the code live automatically. For many developers, this is a scary prospect. What happens if something does not work? What happens if the site or application breaks? What if the code goes out late on Friday, and someone has to work over the weekend?

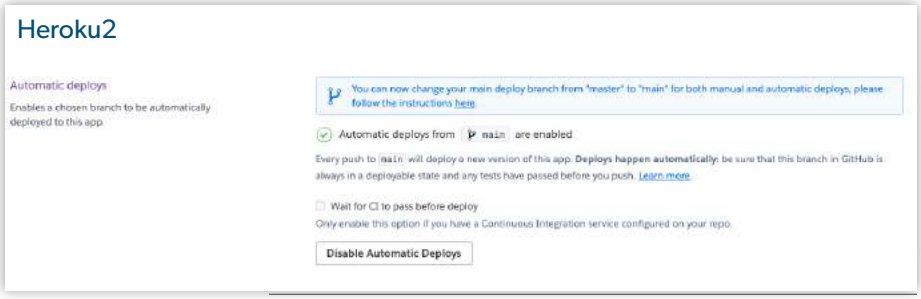
The thing is, these questions are not exclusive to automatically deploying. I have had plenty of times where a manual deploy did not work, or code that passed code reviews and development deploys still had issues in production. All you are doing is removing the human from pushing the button.

This step can be a little hard to show off just because every site's deployment is different. All you will be doing is scripting the steps that you manually do to deploy and then telling some task runner, like GitHub Actions, to run.

In my case, for one project, I use Heroku as the host. Heroku allows you to link a project to a GitHub repository and will automatically deploy code every time code is pushed to a branch, in my case, `main`. Heroku allows you to specify a special file, called `Procfile`, that can point to a setup script.

Since Heroku also handles PHP natively, all I have to do is specify what steps I want to run after the code is cloned and `composer install` is called. For this simple app, I just need to create a `.env` file and then run my database migrations. In my `Procfile`, I just specify a `release` script.

```
// Procfile
web: vendor/bin/heroku-php-apache2 public/
release: ./heroku-deploy.sh
```




```
// heroku-deploy.sh
#!/bin/bash
touch .env
vendor/bin/doctrine-migrations migrate --no-interaction
```

Now, every time I push code to my `main` branch, Heroku takes over and deploys my code. Since I use continuous integration and enforce pull requests on this project, I can, at any point, see that all my tests pass. I am confident that when I merge code into `main` that it is safe to deploy.

While I am using Heroku above, there are plenty of hosts that allow you to deploy code automatically. If you use Laravel Forge[9], you can add a GitHub Action to automatically deploy your code when you commit. At my day job, we have a GitHub action that pushes our GitHub repository automatically to a development repo on our host, which kicks off its own internal deploy. You will need to document how you deploy manually today and figure out the best way to script it.

It's About Confidence

The idea of Continuous Integration and Continuous Deployment is nothing new, but the ability for developers to push code live directly to users has taken the original idea to new heights. By making sure that your code is maintainable and testable, you can work your way to putting all the technical pieces in place. Then you just need to get confidence in your code.

If you are in a position to do Continuous Delivery, I say go for it. It can take a load off of you mentally to know that you feel safe when code goes live. At the very least, implementing Continuous Integration can help show that your code is healthy. Even if you continue to push that deploy button, you can feel safe that everything should work.

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

8 *CircleCI: https://circleci.com*

9 *Laravel Forge: https://phpa.me/laravel-forge-git-actions*

Classifying Ransomware

Eric Mann

One of the terrifying new developments in technology is the high prevalence of ransomware—criminals using software to hold your data or information systems hostage.

Last spring[1], cybercriminals managed to take down the largest fuel pipeline in the United States. This outage triggered fuel shortages and price increases across the East Coast when the Colonial Pipeline was forced to shut down entirely. The outage lasted five days and resulted in Colonial Pipeline paying the criminals over $4 million to restore access to their system.

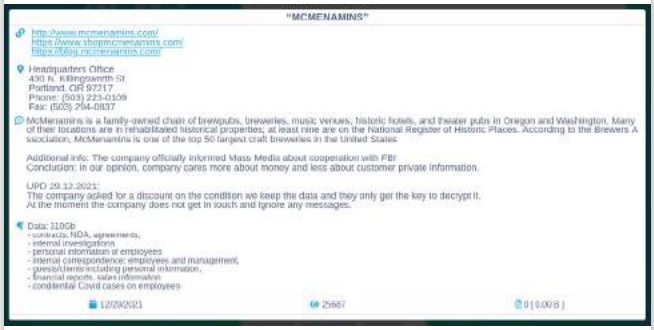
But this isn't the end of the story.

Late last year[2], Portland-based McMenamins also suffered a ransomware attack. Unlike Colonial Pipeline, McMenamins runs a chain of brewpubs and hotels in the Pacific Northwest. Also, unlike previous attacks, this one didn't shut down the company. Instead, the cybercriminals copied sensitive data from McMenamins' systems and threatened to publicly release it if their ransom was not paid.

As customer data did not appear to be breached, McMenamins ultimately chose *not* to pay the ransom. They did, however, offer identity protection services to employees[3] whose data might have been compromised.

McMenamins' situation demonstrates a different form of ransomware than the one that impacted Colonial Pipeline. Colonial's systems were completely locked and shut down, with **restored access** being offered at a price. McMenamins' data was stolen with **its continued secrecy** being offered by the criminals.

Public breach notification as published by the ransomware gang that attacked McMenamins.



One variant of MrLocker ransomware threatened to delete your files. The other merely locked your machine and prevented legitimate access.

Each of these situations demonstrates a nuanced difference between the types of ransomware in the wild today. In general, there are four classes of ransomware of which we need to be aware and against which we need to protect our systems.

Type 1 ransomware

The obvious first class of ransomware is that which hit Colonial Pipeline's systems. This family of attacks seizes control of a system and demands payment for that control to be returned. It could be as simple as a remote changing of administrative passwords.

This family of attacks is commonly called "locker ransomware" because of the utility that locks legitimate users out of a system. An impacted machine displays a pop-up demanding the user input a key to unlock their machine and restore access.

A common example is the MrLocker family of scareware viruses. This family of ransomware was discovered in 2017[4], and it *doesn't* encrypt the files on your machine. However, it does trigger a pop-up demanding a payment in Bitcoin "or else."

One variant of the ransomware claimed your files would be deleted if you failed to pay. Another variant would merely block access until you paid and received an unlock code. In either case, this form of ransomware prevents legitimate access to a computer system until such ransom is paid.

Type 2 ransomware

When the talking heads on TV speak about ransomware, they're usually talking about the form of ransomware that encrypts the data on your machine and demands payment for

1 *Last spring:* https://phpa.me/bloomberg-colonial-breach

2 *Late last year:* https://phpa.me/bleepingcomputer-mcmenamins

3 *identity protection services to employees:* https://lifa.rs/3ok4ti3

4 *discovered in 2017:* https://twitter.com/malwrhunterteam/status/871709340473982976

Example of the WannaCry encryption splash screen.

the decryption key. The most familiar names in this realm are CryptoLocker[5], WannaCry[6], and Petya[7].

These malware attacks targeted Windows-based systems using well-studied vulnerabilities that allow them to spread laterally across networks and quickly encrypt the base filesystem of the computer. They'd demand payment in Bitcoin with individually-trackable wallet addresses in exchange for a unique decryption key to restore system access.

Luckily, the WannaCry malware attacks only lasted a few hours as a security researcher quickly discovered and triggered a kill switch that disabled the malware globally. Even with that quick response, WannaCry alone likely caused more than $4 billion[8] in damage.

Type 3 ransomware

One of the most recent families of ransomware is straight-up extortion. Criminal gangs are working to breach large corporations by way of insecure employee accounts. They then leverage their access to download as much private information as possible before threatening to post it online.

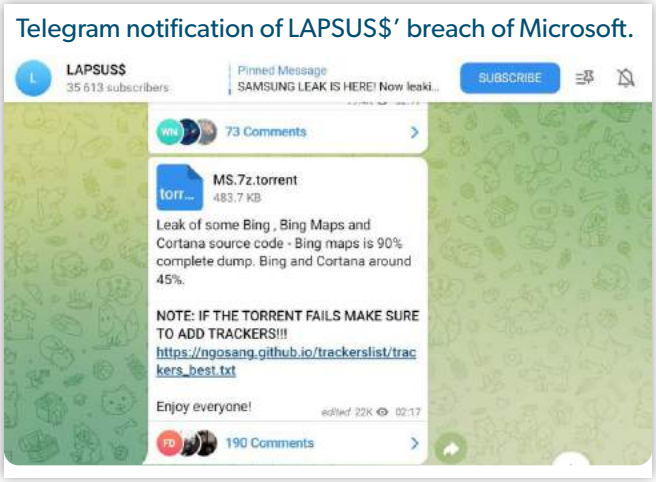
One of these gangs is called Conti—they were the ones who hit McMenamins and multiple companies[9] in the financial and energy sectors. In most cases, they post a small sample of files on a site on the darkweb, followed by a direct notification to the hacked company. If the company fails to pay, they start leaking more and more documentation.

Another such gang calls itself LAPSUS$ and communicates via Telegram. In March, LAPSUS$ managed to breach Nvidia[10], Microsoft, and even the identity provider Okta[11]! In every case, the attackers made their way into the system via weak passwords used by employees. They then moved laterally throughout the system to exfiltrate as much sensitive information as possible prior to detection.

Type 4 ransomware

The most terrifying form of ransomware is also the most chaotic. In 2017, a new variant of Petya was discovered and called, confusingly, "NotPetya[12]." This variant used a vulnerability in Windows similar to that used by WannaCry in order to spread across networks.

Telegram notification of LAPSUS$' breach of Microsoft.



However, unlike files encrypted by WannaCry or Petya, files encrypted by NotPetya were *not* recoverable. Not even with the password provided by the original hackers!

It was quickly determined that NotPetya wasn't being used by hackers to raise money. The demanded ransom was relatively low (similar to the earlier MrLocker family of scareware). But the damage it could do to a system was massive. Worldwide, the NotPetya ransomware variant likely caused several billion dollars worth of damage, but never actually ransomed any recoverable data.

The sky is not falling ...

Ransomware is the stuff of nightmares, and, honestly, it keeps me up most nights. But it's not the end of the world. You can do several things to properly protect yourself from all four flavors of attack.

First, ensure you have reliable, offsite backups of critical data and business code. Ideally, you should be able to restore regular business operations from these backups alone (i.e., if your existing production environment entirely disappeared).

5 *CryptoLocker:* https://en.wikipedia.org/wiki/CryptoLocker

6 *WannaCry:* https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

7 *Petya:* https://en.wikipedia.org/wiki/Petya_and_NotPetya

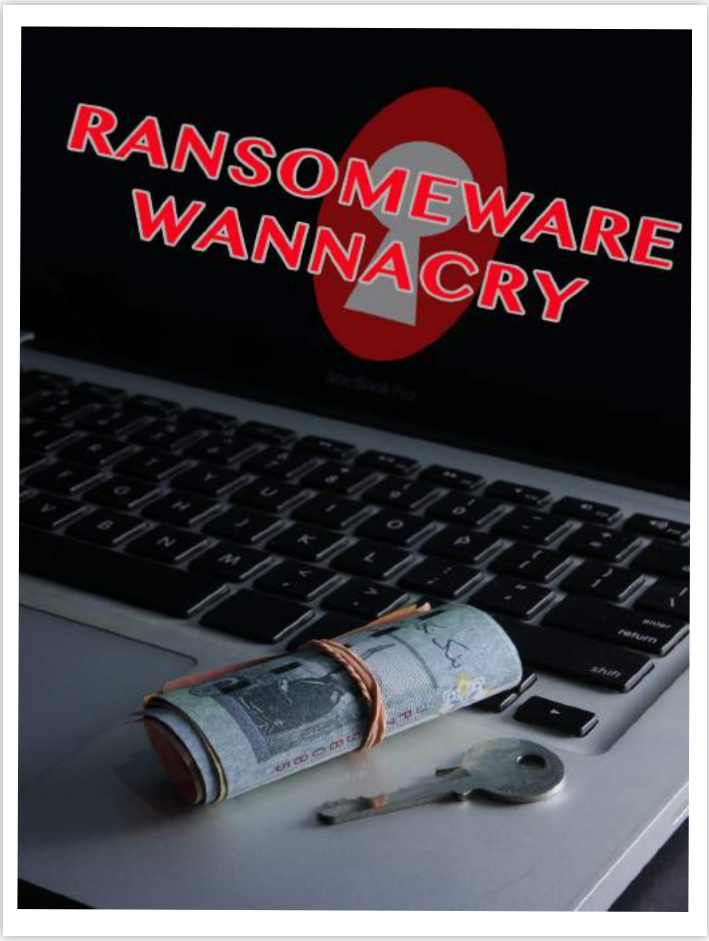
8 *more than $4 billion:* https://securityintelligence.com/?p=431109

9 *multiple companies:* https://phpa.me/esentire-conti-companies

10 *Nvidia:* https://phpa.me/wired-lapsus-nvidia

11 *Okta:* https://phpa.me/okta-lapsus

12 *NotPetya:* https://www.hypr.com/notpetya/



Take time to test your backups regularly to ensure they're stable and reliable.

Second, configure strong anomaly monitoring in production. AWS customers can leverage tools like GuardDuty[13] to automatically scan for and alert on common threats. It's important that your team detect and remediate any potential intrusion early to prevent the kinds of leaks caused by type 3 ransomware.

Next, install a strong anti-malware system on all of your system endpoints—meaning employees' machines. The easiest way to protect from a ransomware infection starting on an end-user's machine is to scan for and block infections on their machines.

Finally, take a deep breath. The barbarians truly are at the gate, but we have the tools necessary to keep our data backed up, monitor for potential breaches, and stop them before they get a foothold. With any luck, learning the various classifications of ransomware will help protect you from them *without* costing you any sleep.

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

13 *GuardDuty:* https://aws.amazon.com/guardduty/

New and Noteworthy

PHP Releases

PHP 8.1.5 (Released!):

https://www.php.net/archive/2022.php#2022-04-15-1

PHP 8.0.18 (Released!):

https://www.php.net/archive/2022.php#2022-04-14-2

PHP 7.4.29 (Released!):

https://www.php.net/archive/2022.php#2022-04-14-1

News

PHP 8.2 Release Manager Selection

With the first alpha of PHP 8.2 due in 6 weeks (unless we are going to change our usual schedule), I think it's time to start the process of finding and electing release managers for the next minor release of PHP.

https://externals.io/message/117595

New in Symfony 6.1: Improved Routing Requirements and UTF-8 Parameters

Using PHP BackendEnum as route parameters, a collection of common routing, and UTF-8 Parameter Name.

https://phpa.me/symfony-6-1-routing-utf8-parameters

PHP Foundation Core Developers

Since PHP Foundation was publically announced, we were all curious about who will work full time on developing the language, how the foundation will be structured, who will have the decision power, and other aspects related to the future of the language.

https://phpa.me/laravelmagazine-foundation-core-developers

Create PHP Courses Inside PhpStorm With EduTools

At some point in your professional career, you may want to share your programming knowledge with your co-workers or students. You may even want to start your own blog, YouTube channel, or an account on social media to share your experience with a larger audience. If any of this is something you are interested in, we have good news for you. Now you can create your own educational courses inside PhpStorm!

https://blog.jetbrains.com/?p=241750/

While I wasn't paying attention, PHP got quite good

The last time I used PHP was probably around 2017, although it was just in the context of supporting some WordPress sites. By that time 7.2 had already been released, but I had no idea. I wanted to avoid working with PHP at all costs.

https://dnlytras.com/blog/modern-php/

CVE-2022-24828: Composer Command Injection Vulnerability

Please immediately update Composer to version 2.3.5, 2.2.12, or 1.10.26 (composer.phar self-update). The new releases include fixes for a command injection security vulnerability (CVE-2022-24828) reported by Thomas Chauchefoin from SonarSource.

https://phpa.me/composer-cve

Video: PHP Isn't Dead

From the cast of "Laravel Origins: The Documentary", a group of well known developers talk about some of the reasons that developers ourside of the PHP ecosphere think that PHP is dead, and they give their own insights into the reasons it's not.

https://phpa.me/php-isnt-dead

Get Organized and Get Started

Edward Barnard

This month we begin the preliminary "spadework" to set up our project. We'll be taking a hands-on "code first" approach to implementing Strategic Domain-Driven Design.

All source code is available on GitHub at ewbarnard/strategic-ddd[1].

Essential Questions

Upon surviving this month's article, you should be able to answer (see Figure 2):

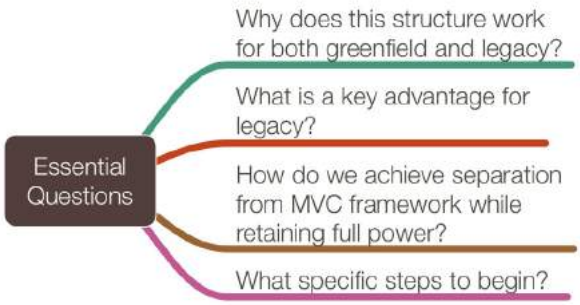
- Why does this way of structuring our codebase work for both new (greenfield[2] projects) and legacy codebases?
- What is a key advantage for our legacy codebase?
- How do we achieve code separation from the MVC framework while retaining the full power of that MVC framework?
- What specific steps do we take to begin the example project?

Figure 1.

Hands-On

Strategic Domain-Driven Design is about communication and collaboration. But once you have strategized, communicated, and collaborated, where do you put the code you write? Beginning this month, we're addressing that need head-on.

Figure 2.



I'm taking a "code first" approach, and here's why. There's been a strong tendency for developers to jump straight to the "tactical" patterns of Domain-Driven Design because they're concrete and relatively straightforward to implement. The abstract concepts such as "Bounded Context" are much less obvious.

Author Nick Tune, for an event session[3] with the Agile Alliance, explains:

> *Many people think that DDD is about software design patterns, but that's only a small part, and the least important part of DDD. In fact, Eric Evans wishes he'd focused more on the strategic aspects of DDD in his famous book and pushed the tactical coding patterns to the back!*

When first learning about Domain-Driven Design, we learn that when we get to writing the actual code, the code should fit inside one or more Bounded Contexts. The Bounded Context is one of Evans' strategic aspects.

What is a Bounded Context, and where should we draw the boundary? The answer is, "it depends," which is not terribly helpful when we actually need to start implementing a feature. The problem is in figuring out where to start.

When starting, we, therefore, need to address the combined problems:

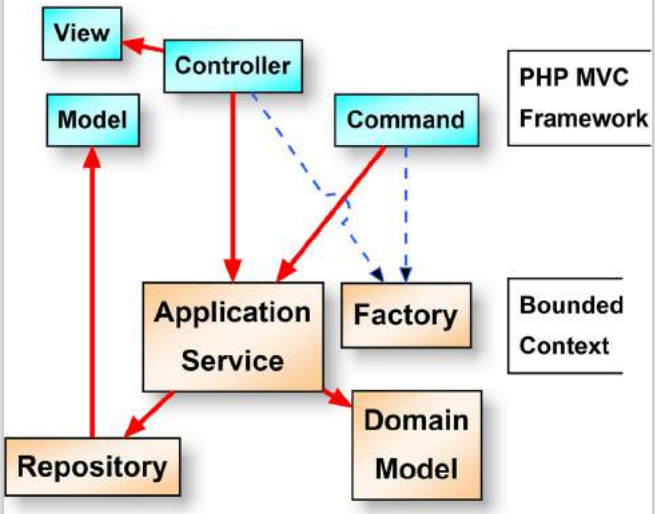
- Analysis paralysis in the sense of trying to figure out how and where to start
- The sharp learning curve involved in learning Domain-Driven Design

1 *ewbarnard/strategic-ddd:* https://github.com/ewbarnard/strategic-ddd

2 *greenfield:* https:///wiki/Greenfield_project

3 *event session:* https://phpa.me/agilealliance-strategic-ddd

Figure 3. Bounded Context Design

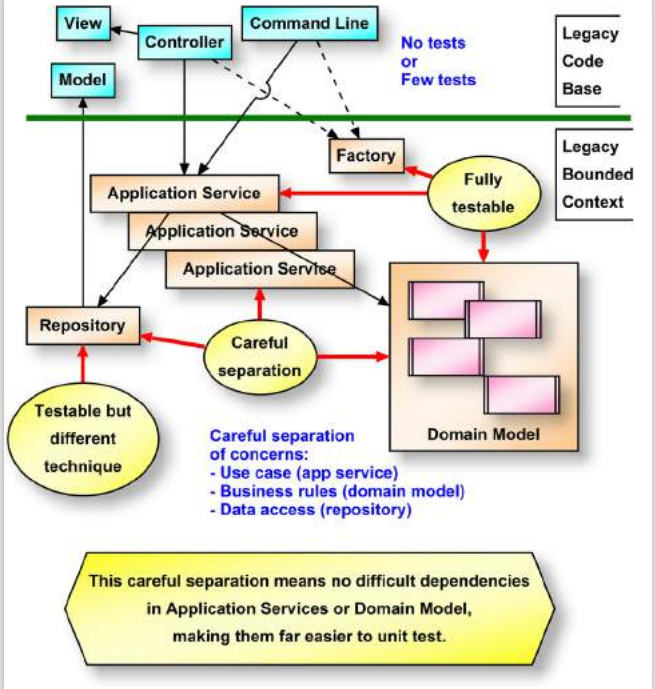


That's why we're here–to get organized and get started with Strategic Domain-Driven Design. I want you to immediately begin gaining your own real-world experience with your own projects.

As Evans himself confirms, it's not the tactical coding patterns that are important. It's the bigger picture, what Evans calls the strategic aspects, that brings the value. We'll see these strategic aspects in example after example. That's where you will gain the deeper insight foundational to DDD.

I will be quoting other authors many times. It's not because I'm unable to think for myself. Rather, it's demonstrating how you can integrate other peoples' knowledge and experience into your own workflow.

Figure 4. Legacy Code Base



I am purposely *not* organizing "DDD Alley" around those published DDD design patterns. I see PHP-based development as pragmatic and results-focused. Rather than diving deeply into what a particular pattern *is*, I prefer to show you how I *use* that pattern in a real-world setting. We'll be explicitly using pattern after pattern, thus gaining the benefit of other peoples' experience in a hands-on manner.

Strategic Domain-Driven Design, I have found, proves to be an excellent fit for this pragmatic approach to getting our projects built. Figure 3 shows how we will be structuring each Bounded Context and its relation to whatever PHP framework we're using.

Legacy Code Base

With this project, we'll be coding from scratch. It wouldn't make much sense to show you a production codebase because it's not *your* production codebase. Instead, we'll start with an empty repository and focus on the essentials.

This structure *does* work with a legacy codebase. That's the whole point! See Figure 4.

One particularly valuable outcome will be that the legacy codebase becomes testable (if it isn't already). As new features become part of a Bounded Context, we'll design them for testability. The same would be true for existing functionality that needs to be reworked or rewritten. If it's rewritten as part of the Bounded Context, that should mean it's being reworked to include testability.

Note, however, that the Bounded Context is the destination for *new* material. We are *not* proposing to add tests to existing code. That's a whole separate topic of its own.

Michael Feathers, in *Working Effectively with Legacy Code*, page 77, laments:

> *How long does it take to make changes? The answer varies widely. On projects with terribly unclear code, many changes take a long time. We have to hunt through the code, understand all the ramifications of a change, and then make the change. In clearer areas of the code, this can be very quick, but in really tangled areas, it can take a very long time. Some teams have it far worse than others. For them, even the simplest code changes take a long time to implement.*

Our objective, as we implement new features or use cases within a legacy codebase, is to make future changes easier and easier. Note that Figure 4 was not merely a restatement of Figure 3. Figure 4 focuses on testability. Why is that?

Feathers explains the dilemma (pp. 14-15):

> *So how do we start making changes in a legacy project? The first thing to notice is that, given a choice, it is always safer to have tests around the changes that we make. When we change code, we can introduce errors; after*

> *all, we're all human. But when we cover our code with tests before we change it, we're more likely to catch any mistakes that we make.*

Unfortunately, it's not that simple. Feathers continues on page 16:

> *All of these problems are dependency problems. When classes depend directly on things that are hard to use in a test, they are hard to modify and hard to work with. Dependency is one of the most critical problems in software development. Much legacy code work involves breaking dependencies so that change can be easier.*
>
> *The legacy code dilemma: When we change code, we should have tests in place. To put tests in place, we often have to change code.*

We will not be introducing tests to legacy code. But it's useful to understand why legacy code is so difficult to work with so that as we write new code, we can aim to avoid the same trap.

Martin Fowler sums up our situation when explaining his Strangler Fig Application[4] pattern:

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

Secret Feature

Our first feature is what I jokingly call a secret feature because, for me, it's more fun to think of it that way. Seriously, though, sometimes it's hard to get tasks into the development cycle when all they do is increase code quality. All too often, if the feature doesn't provide visible benefit to the end-user, it doesn't make the sprint.

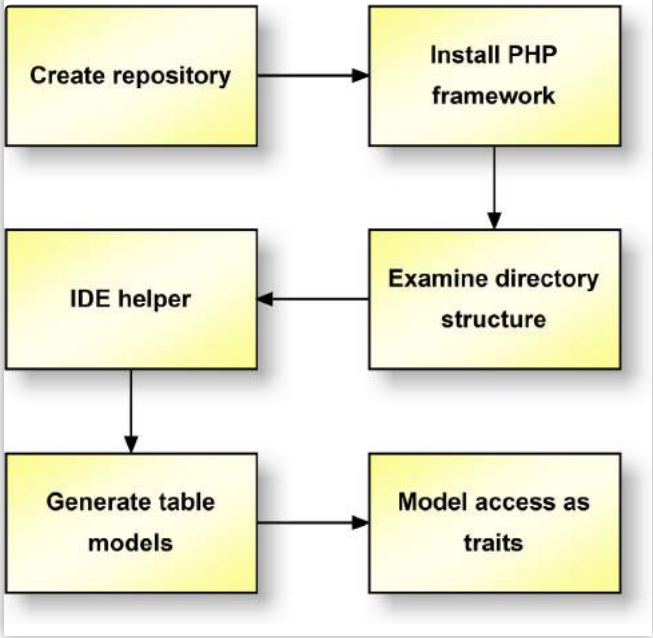
What is this secret feature? It aims to help answer the question "what went wrong?" when certain types of failures occur. I'll explain the feature next month with "Random and Rare Failures." This month we're taking on the preliminary spadework[5] of getting our PHP project set up.

We'll be taking steps to separate our code development from the framework's Model View Controller (MVC) conventions while still using the full power of that PHP framework. Let's get started. Figure 5 shows our preliminary project setup steps.

4 *Strangler Fig Application:* https://martinfowler.com/bliki/StranglerFigApplication.html

5 *spadework:* https://www.merriam-webster.com/dictionary/spadework

Figure 5. Project Setup Steps



Project Repository

I use CakePHP[6] as my preferred PHP framework. The framework does not matter; what's important is making the distinction between what code stays within the framework and what code steps outside the framework.

I have placed our project repository on GitHub as ewbarnard/strategic-ddd[7]. The basic CakePHP project is installed in branch b001-create-project[8]. Look inside the top-level folder strategic_ddd[9] to see the freshly-installed CakePHP project.

You are welcome to clone or fork the repository. The CakePHP[10] installation guide should get you up and running.

- You'll need to create and edit the configuration file based on the sample. Your database connection credentials should be the only thing needed for now.
- You'll need to run `composer install` to bring in the project dependencies.

Note the top-level folders `bin/`, `src/`, `tests/`. Nearly all code we'll be writing will go in `src/` or, for automated tests, `tests/`. The tools that come as part of CakePHP are in `bin/`.

The src[11] folder contains:

- `Controller/`
- `Model/`
- `View/`

6 *CakePHP:* https://cakephp.org

7 *ewbarnard/strategic-ddd:* https://github.com/ewbarnard/strategic-ddd

8 *b001-create-project:* https://github.com/ewbarnard/strategic-ddd/tree/b001-create-project

9 *strategic_ddd:* https://phpa.me/github-ew-strategic-ddd

10 *CakePHP:* https://cakephp.org

11 *src:* https://phpa.me/github-ew-ddd-src

Listing 1.

```
# Documentation:
# https://github.com/dereuromark/cakephp-ide-helper

# Installation:
# https://github.com/dereuromark/cakephp-ide-helper/tree/master/docs

# Install:
composer require --dev dereuromark/cakephp-ide-helper
bin/cake plugin load IdeHelper

# Use:

bin/cake illuminator illuminate -v
bin/cake annotate all -v
bin/cake code_completion generate
bin/cake phpstorm generate
```

Listing 2.

```
CREATE TABLE `phinxlog`
(
    `version`        bigint     NOT NULL,
    `migration_name` varchar(100)        DEFAULT NULL,
    `start_time`     timestamp  NULL     DEFAULT NULL,
    `end_time`       timestamp  NULL     DEFAULT NULL,
    `breakpoint`     tinyint(1) NOT NULL DEFAULT '0',
    PRIMARY KEY (`version`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb3;
```

These are for the portions supporting the Model View Controller (MVC) structure. `src/` also contains the folder `Console/`, which we'll ignore, and `Application.php`, which may need to be edited for adding plugins, authentication middleware, etc.

The code we're about to write is in the next branch, b002-transaction-failures[12].

IDE Helper

Since I use CakePHP, I also use Mark Scherer's CakePHP IdeHelper Plugin[13]. The installation and usage instructions are at the bottom of that GitHub page.

This installation is optional. We'll be using class constants generated by the IdeHelper, but it will be clear from the context that those constants only matter if you're also using CakePHP in your own project.

Listing 1 shows the steps for installation and usage. Use the `--dev` switch with `composer require` because the IdeHelper does not need to run in production. Scherer recommends (in the Setup section of the installation instructions) adding an environment check when loading the plugin to ensure it does not get loaded in production.

The CakePHP IDE Helper takes advantage of Phinx migrations[14]. Table `phinxlog` will appear in your database as shown in Listing 2.

Generate Models

CakePHP is able to generate its own model classes by introspecting the MySQL database. In keeping with the "cake" theme, CakePHP calls this "baking" and provides a "bake" command for the purpose. Here is the preliminary script to rebuild all CakePHP model classes, including enhancements coming from the CakePHP IDE Helper plugin. See Listing 3.

Listing 3.

```
#!/bin/bash -xv
bin/cake cache clear_all

BAKE="bin/cake bake model -f --no-fixture --no-test"

TABLES_PRIMARY="domain_events \
event_counts \
local_app_events"

# shellcheck disable=SC2034
for pass in 1 2; do
    for j in $TABLES_PRIMARY; do
        echo baking "$j"
        $BAKE "$j"
    done
done

bin/cake illuminator illuminate -v
bin/cake annotate all -v
bin/cake code_completion generate
bin/cake phpstorm generate

echo
echo "$0: All done."
echo
```

The model-generation script "bakes" all models twice. I generate the models twice because the models incorporate relationships between tables. If the relationship between tables A and B changes, when generating A, the change won't be picked up because B has not yet been regenerated. On the second pass, both A and B pick up changes established in the first pass.

Model Access as Trait

CakePHP provides automatic access to its database layer via its Model classes. We are not here to throw away the framework. We definitely want to use its power (including security-related features). However, with Strategic

12 *b002-transaction-failures:* https://phpa.me/github-ew-ddd-failures

13 *CakePHP IdeHelper Plugin:* https://github.com/dereuromark/cakephp-ide-helper

14 *Phinx migrations:* https://book.cakephp.org/phinx/0/en/index.html

Listing 4.

```
...
 5. namespace App\BoundedContexts\Infrastructure\LoadTableModels\PrimaryDatabase\Events;
 6.
 7. use App\Model\Table\LocalAppEventsTable;
 8. use Cake\ORM\Locator\LocatorAwareTrait;
 9.
10. trait AppEventsPrimaryTrait
11. {
12.     use LocatorAwareTrait;
13.
14.     protected LocalAppEventsTable $localAppEventsTable;
15.
16.     protected function loadLocalAppEventsTable(): void
17.     {
18.         $this->localAppEventsTable = $this->localAppEventsTable();
19.     }
20.
21.     protected function localAppEventsTable(): LocalAppEventsTable
22.     {
23.         /** @noinspection PhpUnnecessaryLocalVariableInspection */
24.         /** @var LocalAppEventsTable $table */
25.         $table = $this->getTableLocator()->get('LocalAppEvents');
26.         return $table;
27.     }
28. }
```

Domain-Driven Design, we'll be writing much of our code outside the framework's structure. We, therefore, need to do a bit extra to gain access to CakePHP's database layer.

We are separating that "a bit extra" into a set of PHP traits, with one trait for each cluster of database tables. Listing 4 shows our first such trait. This "cluster" of table models only includes one model.

Listing 5 is similar. This trait provides access to the `event_counts` table. CakePHP, by convention, constructs "camel case" Model class names from the "snake case" table name.

Listing 5.

```
...
 5. namespace App\BoundedContexts\Infrastructure\LoadTableModels\PrimaryDatabase\Events;
 6.
 7. use App\Model\Table\EventCountsTable;
 8. use Cake\ORM\Locator\LocatorAwareTrait;
 9.
10. trait EventCountsTrait
11. {
12.     use LocatorAwareTrait;
13.
14.     protected EventCountsTable $eventCountsTable;
15.
16.     protected function loadEventCountsTable(): void
17.     {
18.         $this->eventCountsTable = $this->eventCountsTable();
19.     }
20.
21.     protected function eventCountsTable(): EventCountsTable
22.     {
23.         /** @noinspection PhpUnnecessaryLocalVariableInspection */
24.         /** @var EventCountsTable $table */
25.         $table = $this->getTableLocator()->get('EventCounts');
26.         return $table;
27.     }
28. }
```

Listing 6.

```php
<?php

declare(strict_types=1);

namespace App\BoundedContexts\Infrastructure\LoadTableModels\PrimaryDatabase\Events;

use App\Model\Table\DomainEventsTable;
use Cake\ORM\Locator\LocatorAwareTrait;

trait GlobalEventsTrait
{
    use LocatorAwareTrait;

    protected DomainEventsTable $domainEventsTable;

    protected function loadDomainEventsTable(): void
    {
        $this->domainEventsTable = $this->domainEventsTable();
    }

    protected function domainEventsTable(): DomainEventsTable
    {
        /** @noinspection PhpUnnecessaryLocalVariableInspection */
        /** @var DomainEventsTable $table */
        $table = $this->getTableLocator()->get('DomainEvents');
        return $table;
    }
}
```

Listing 6 is again similar. We have not yet seen the tables themselves, but we'll be using these traits to gain access.

When using the trait, each table model becomes a protected property of that class. Since there's no need to run through the relatively slow initialization for every table model on every web request, I expect the class to call the `loadXXX()` method for those table models it will use.

Legacy Setup

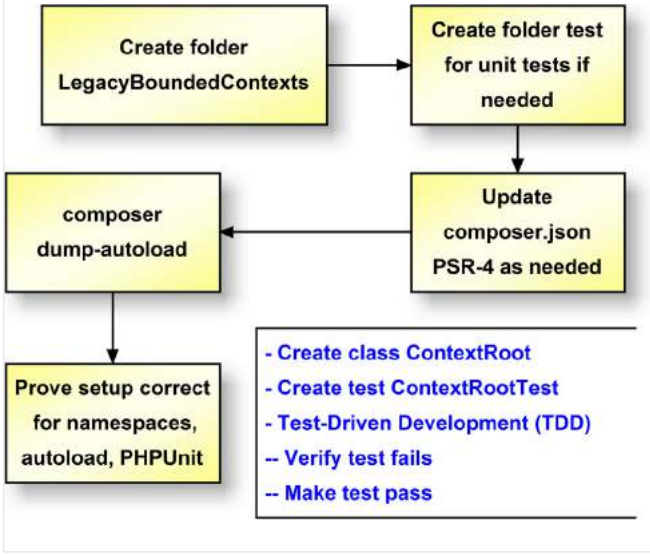
One advantage of Strategic Domain-Driven Design in PHP is that we can implement the concepts in a legacy codebase just as well as with a new project. In fact, this is more likely to be your own situation, given that it's far more common (and generally wiser) to continue with an existing codebase.

The legacy setup steps are minimal (Figure 6):

- Create a Bounded Contexts folder.
- Create a folder for unit tests (if it does not already exist).
- Update the composer namespace information.
- Write a test to verify the setup is operational.

Create a top-level Bounded Contexts folder. Since we'll be seeing listings from both the example project and from my legacy codebase, I am naming this folder

Figure 6.



`LegacyBoundedContexts` to distinguish it from the example project's `BoundedContexts`. Create a class inside the folder to ensure PSR-4 autoloading[15] is working correctly. See Listing 7.

Listing 7.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace LegacyBoundedContexts;
6.
7. class ContextRoot
8. {
9.     public function echoBack(string $input): string
10.    {
11.        return '';
12.    }
13. }
```

Create the folder and namespace structures for unit testing with PHPUnit[16]. These things may already exist and be set up in your legacy codebase. Create a test that fails. This test aims to ensure your setup, autoloading, and test setup is operational and running as expected. See Listing 8.

The test should fail as shown below.

```
PHPUnit 7.5.20 by Sebastian Bergmann and contributors.

Failed asserting that two strings are identical.
Expected :'test input string'
Actual   :''
<Click to see difference>

Time: 193 ms, Memory: 12.00 MB

FAILURES!
Tests: 17, Assertions: 27, Failures: 1.
Process finished with exit code 1
```

Note that the legacy codebase is running an outdated version of PHPUnit. (PHPUnit 7 was released in early 2018.) You may have a similar situation where tools are locked to old versions in support of that codebase.

This step is important. If you don't see the expected failure message, that probably means your setup is not correct. Once your test is failing as expected, update the system under test (`class ContextRoot`) so that the test passes. See Listing 9.

All tests should now run green. See Figure 7. The phrase "running green" comes from the test runner. PHPUnit normally colors the output green to show passing tests.

15 *PSR-4 autoloading:* https://www.php-fig.org/psr/psr-4/

16 *PHPUnit:* https://phpunit.de/

Listing 8.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace Test\LegacyBoundedContexts;
6.
7. use LegacyBoundedContexts\ContextRoot;
8. use PHPUnit\Framework\TestCase;
9.
10. class ContextRootTest extends TestCase
11. {
12.     public function testEchoBack(): void
13.     {
14.         $target = new ContextRoot();
15.         $expected = 'test input string';
16.         $actual = $target->echoBack($expected);
17.
18.         static::assertSame($expected, $actual);
19.     }
20. }
```

Listing 9.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace LegacyBoundedContexts;
6.
7. final class ContextRoot
8. {
9.     public function echoBack(string $input): string
10.    {
11.        return $input;
12.    }
13. }
```

Figure 7.

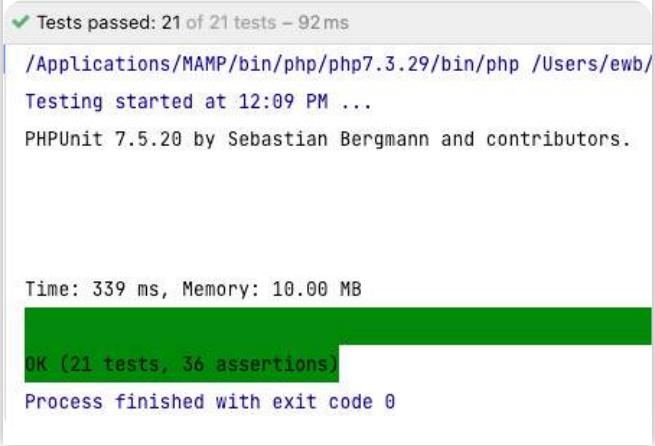
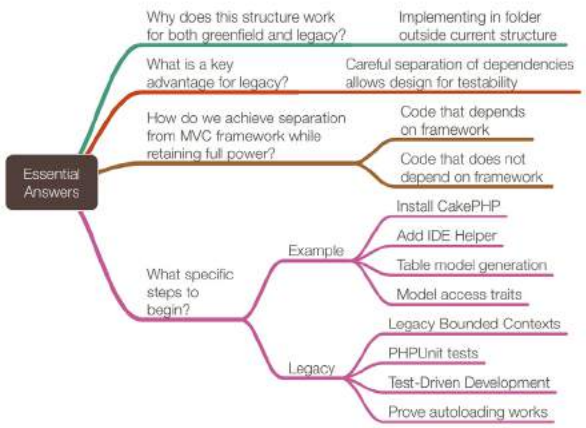


Figure 8. Essential Questions Answered



Essential Questions Answered

- *Why does this way of structuring our codebase work for both new greenfield projects and legacy codebases?* We're implementing Strategic Domain-Driven Design in a folder or folders sitting outside existing structures (libraries and frameworks).
- *What is a key advantage for our legacy codebase?* The careful separation of dependencies allows us to design new and updated code for testability.
- *How do we achieve code separation from the MVC framework while retaining the full power of that MVC framework?* We carefully separate parts of the code that do not depend on the framework from parts of the code that do depend on the framework.
- *What specific steps do we take to begin the example project?* We installed a default CakePHP project, added the IDE helper, created a script for generating table models, and set up PHP `trait` files to simplify CakePHP model access from our code sitting outside the CakePHP framework. We set up a Legacy Bounded Contexts folder for the legacy codebase and wrote a unit test ensuring the setup and autoloading are operational.

Summary

There's been a strong tendency for developers to jump directly to the tactical design patterns of DDD, leaving full Strategic Domain-Driven Design for later. That's a mistake and indeed misses the whole point of moving to DDD.

The problem, in my view, has been in figuring out where to start. Therefore we are beginning a "code first" approach, showing exactly how and where to start.

1. We created a new CakePHP project.
2. All source code is available from GitHub repository ewbarnard/strategic-ddd[17].
3. We generated the CakePHP-specific table models.
4. We built a series of PHP traits for later use in accessing those table models from outside the CakePHP framework.

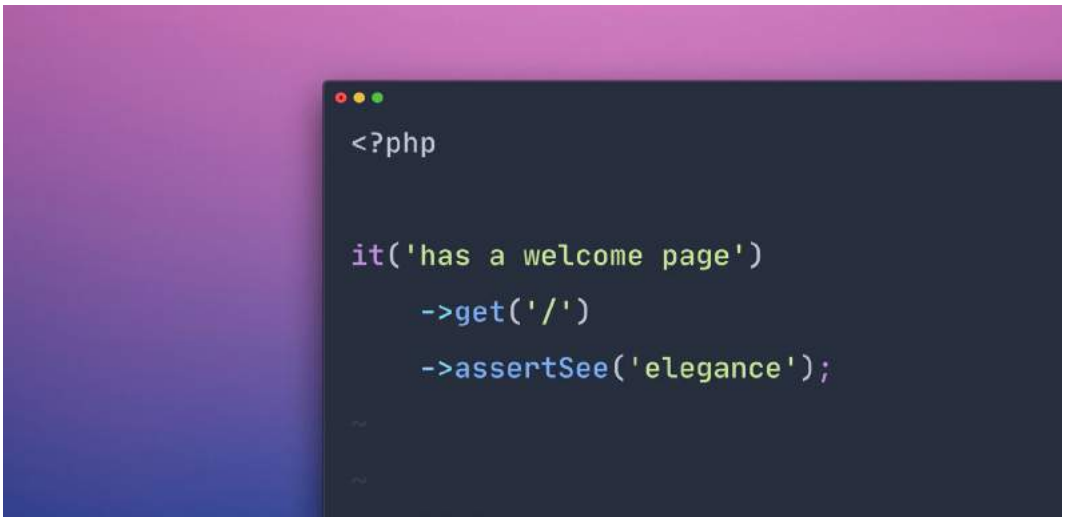
Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching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

17 ewbarnard/strategic-ddd: https://github.com/ewbarnard/strategic-ddd

Pest Control

Marian Pop

PEST is a PHP testing framework built on top of PHPUnit that offers a functional approach to writing tests, eliminating as much boilerplate as possible and focusing on the tests themselves. It was created by Nuno Maduro (Laravel Core Team member) in 2020, and since then, it has had 1.6M downloads.



PEST is framework agnostic, and it can be used in any PHP project; however, in this article, we will be focusing on how you can use PEST in a Laravel application.

Installation

Getting up and running with PEST is as simple as running 3 commands in your terminal. First, we have to require PEST via composer:

```
composer require pestphp/pest --dev
--with-all-dependencies
```

Then, we have to require the PEST Laravel plugin:

```
composer require pestphp/pest-plugin-laravel --dev
```

Finally, you have to run the PEST `artisan` install command:

```
php artisan pest:install
```

Great. You can now run `./vendor/bin/pest` in your terminal. PEST will run your existing PHPUnit tests because, being built on top of PHPUnit, both PHPUnit test classes and PEST test files can coexist in the same test suite. You can also reuse your Test Case classes in PEST test files.

Features

Let's take a look at what PEST offers in terms of features and how we can reduce PHPUnit's boilerplate and simplify our tests, making them more readable.

Start Writing Your First Pest Test

To start writing PEST tests is as simple as creating a new file in the `Unit` or `Feature` folder (ex. Test.php). All you need inside this file is a function that runs your test:

```
it('has home', function () {
    // ..
});
```

PEST provides two functions for writing tests: `test()` and `it()`. Use either one, or both, depending on your test naming convention. They behave in the same way and have the same syntax:

```
test('asserts true is true', function () {
    $this->assertTrue(true);

    expect(true)->toBeTrue();
});
```

```
✓ asserts true is true
```

When using `it()`, your test name gets prepended with 'it' in the output:

```
✓ it asserts true is true
```

Expectation Api

PEST offers you a set of expectations[1] in addition to assertions. These functions allow you to compare your values to a set of conditions. Jest was the inspiration for this API. Expectations also enable you to write your tests as though they were natural sentences:

```
test('expect true to be true', function () {
  // assertion
  $this->assertTrue(true);

  // expectation
  expect(true)->toBe(true);
});
```

Higher-order Tests

Higher-order tests, which are shortcuts for performing common activities while creating your tests, are also supported by PEST. The easiest way to think about it is that if you don't supply a closure, the chained methods will do so for you. The simplest basic example is as follows:

```
test('true is true')->assertTrue(true);
```

There's also a small caveat with the higher-order tests and something that I ran into when I wrote my first PEST test: sometimes, you'll need to access methods not available until runtime. To do that you can make use of the tap method provided by the higher-order test which receives a closure and executes the code inside:

```
it('writes to the database after the command is run')
    ->tap(fn() => $this->artisan('your-command'))
    ->assertDatabaseHas('users', ['id' => 1]);
```

1 *expectations:* https://pestphp.com/docs/expectations#available-expectations

Custom Helpers

While PEST comes with a lot of power out of the box, you could have some testing code that is unique to your project that you don't want to duplicate in every test. Custom helpers can help you enhance readability and cut down on code in your test suite. PEST loads the `tests/Pest.php` file by default. You can put your custom helpers in this file. Of course, if the helpers are unique to a test file, you can only use them in that test file.

Consider the following example of a custom helper for the `actingAs` Laravel helper:

Listing 1.

```
1. <?php // tests/Pest.php
2.
3. /**
4.  * Set the currently logged-in user for the application.
5.  *
6.  * @return TestCase
7.  */
8. function actingAs(
9.    Authenticatable $user, string $driver = null
10. ): TestCase {
11.     return test()->actingAs($user, $driver);
12. }
```

Now, you can use the actingAs helper in your tests:

```
<?php

it('redirects to user profile', function () {
    $user = User::factory()->create();

    actingAs($user)
        ->get('/profile')->assertSee($user->name);
});
```

Groups of Tests

The group function in PEST allows you to assign tests to multiple groups. If you have a lot of slow tests, it might be a good idea to put them all in the same group:

```
it('has home', function () {
    // ..
})->group('integration');
```

Alternatively, you can assign a test to multiple groups like so: `->group('integration', 'browser');`. Sometimes, you may want to assign an entire file to a group, and you can do that using the `uses()` method:

```
uses()->group('integration');
```

Finally, you can use the `–group` option to perform the tests of a specified group while executing PEST from the command line:

```
./vendor/bin/pest --group=integration,browser
```

Or you can exclude a group:

```
./vendor/bin/pest --exclude-group=api
```

Elegant Output

One of the features that most newcomers to PEST love are the beautiful and easy-to-read test outputs.

And More...

PEST offers first-party IDE plugins for PhpStorm and VSCode, as well as a VSCode Snippets plugin. You can find them in your IDE's extensions marketplace by typing in "PEST." Besides the IDE plugins, PEST also has 12 first-party plugins that allow you to extend its functionality. Some examples are `Mock`, `Parallel`, and `Faker`. You can find all of the available plugins on `pestphp.com/docs` under the `plugins` section.

There are more great features we can talk about, but this article is meant to be an introduction to PEST, so we're saving features like `Datasets`, `Coverage`, `CLI Options`, and `Parallel Testing` for future articles.

Marian Pop is a PHP / Laravel Developer based in Transylvania. He writes and maintains LaravelMagazine.com and hosts "The Laravel Magazine Podcast". @mvpopuk

Psr-3 Logger Interface

Frank Wallen

Now that we have introduced the Autoloader (March Issue) and style guide PSRs (March and April), we're going to look at PSR-3[1], the Logger Interface. We'll continue along the path of recommendations for code structure and see recommendations for expectations and behaviors.

Interfaces are a very important part of the communication and interaction between objects and components. They often are referred to as `Contracts`, a term that somewhat captures the objective of the interface. An interface defines the "public" behavior expected of the class implementing the interface. You can read more about them in PHP's Object Interface[2] documentation.

The Logger Interface recommendation requires only 9 methods in the Logger class: *emergency, alert, critical, error, warning, notice, info, debug,* and *log*. Eight of those methods relate to the log levels defined in RFC 5424[3]. The ninth method, *log*, does not relate to a specific level. It expects a log level as its first argument and should behave as defined in the PSR:

> *Calling this method with one of the log level constants MUST have the same result as calling the level-specific method.*

There is a package created and maintained by PHP-fig members on Github[4]. If you are using Composer, it is psr/log.[5] Using this package can ease your efforts when building your own logger and allow other objects to use your logger seamlessly. The **psr/log** package provides a Psr/Log/AbstractLogger that can be extended as show in Listing 1.

The `AbstractLogger` uses a Trait named `LoggerTrait` that implements the other 8 methods, so you don't have to. However, each of those methods calls *this* log function, so it's crucial to implement it! Now the developer can build the pipeline to record and handle logs however they feel is appropriate for their application.

Log Levels

When passing identifying strings around code, defining them as a `const` is recommended. Doing so avoids misspelling

Listing 1.

```php
<?php

namespace App;

use Psr\Log\AbstractLogger;

class Logger extends AbstractLogger
{
    public function log($level, \Stringable|string $message, array $context = []): void
    {
        // TODO: Implement log() method.
    }

}
```

Listing 2.

```php
namespace Psr\Log;

/**
 * Describes log levels.
 */
class LogLevel
{
    const EMERGENCY = 'emergency';
    const ALERT     = 'alert';
    const CRITICAL  = 'critical';
    const ERROR     = 'error';
    const WARNING   = 'warning';
    const NOTICE    = 'notice';
    const INFO      = 'info';
    const DEBUG     = 'debug';
}
```

1 *PSR-3:* https://www.php-fig.org/psr/psr-3

2 *Object Interface:* https://www.php.net/oop5.interfaces

3 *RFC 5424:* https://datatracker.ietf.org/doc/html/rfc5424#page-11

4 *Github:* https://github.com/php-fig/log

5 *psr/log.:* https://packagist.org/packages/psr/log

and confusion. Luckily, Psr/Log/LogLevel has been provided just for that use. See Listing 2.

Message

As noted by the argument type, the `$message` can be a string or a `Stringable` object (that can be cast as a string or has a `__toString()` method). In the message, it MAY use placeholders that SHOULD only be composed of alphanumeric characters, underscore, and period. Placeholders MUST be enclosed within single braces { } with no whitespace. The placeholder{ first_name } is invalid due to the whitespace, it should be {first_name}. If placeholders are allowed by the

```
$logger->log(\Psr\Log\LogLevel::ERROR,
    'Error in user account for {first_name} ....',
    [
        'exception' => ...
        'user_id' => ...
        'first_name' => 'Leslie'
  ]
);
```

logger implementation (some might ignore it), then the value should come from the `$context` array, using the placeholder name as a key:

Then, either in the *log* method or further along that pipeline to actually handle it, the message can be interpolated using the context data, and the user's first name would then replace the placeholder {first_name} with `Leslie`. Why not simply compose the message string with the user's name? For recording or display purposes, it may be important to properly encode or even translate that data, depending greatly upon the needs of the application. It may be necessary to scrub some private data while displaying the log, but keep it when recording it elsewhere.

Context

Commonly, the `$context` array is recorded along with the rest of the log data. Enriching that context with additional data from the environment can prove very useful when sleuthing out a strange error's cause. Additionally, PSR-3 also provides some rules about the context data:

> *Every method accepts an array as context data. This is meant to hold any extraneous information that does not fit well in a string. The array can contain anything. Implementors MUST ensure they treat context data with as much lenience as possible. A given value in the context MUST NOT throw an exception nor raise any php error, warning or notice.*

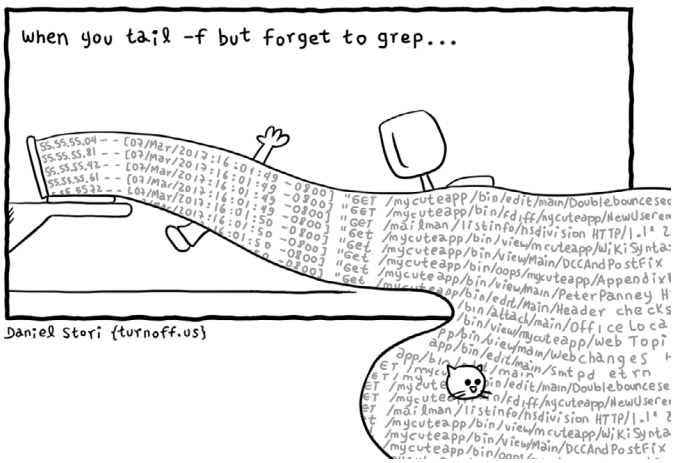
> *If an* `Exception` *object is passed in the context data, it MUST be in the* `'exception'` *key. Logging exceptions is a common pattern and this allows implementors to extract a stack trace from the exception when the log backend supports it. Implementors MUST still verify that the* `'exception'` *key is actually an* `Exception` *before using it as such, as it MAY contain anything.*

Essentially, these rules are saying to keep the context array data clean and 'unsurprising,' let the log handlers do their job, especially in regards to escaping data, as it may not be known in what context the data will be handled.

Conclusion

Almost all modern frameworks come with out-of-the-box implementations for logging systems, often using other popular and road-tested logging packages (like **monolog/monolog** at Packagist[6] . Therefore many developers have not had the need or requirement to write their own logger. Understanding the interface and expectations of a logging system will work as a guide or roadmap when extending an existing logger or even customizing it.

Frank Wallen is a PHP developer and tabletop gaming geek (really a gamer geek in general). He is also the proud father of two amazing young men and grandfather to two beautiful children who light up his life. He lives in Southern California and hopes to one day have a cat again. @frank_wallen



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

6 *Packagist:* https://packagist.org/packages/monolog/monolog

Controlled Randomness

Oscar Merida

Last month's puzzle challenged you to "harness randomness such that we can guarantee that a given input will produce the same random sequence." We'll look at one possible solution and build on it to write the underpinnings of one popular type of board, and now online, game.

Random Number Generators

We usually think of computers as deterministic machines. Given a particular set of inputs, running them through our code should always produce the same, predictable output. But then, how do you simulate the flip of a coin, the toss of a die, or pick a random number between 1 and 100. Crucially, how do you execute each such that one outcome does not depend on the one before it? How do you emulate "randomness" such that an observer can not discern a pattern in your approach?

In some cases, notable non-cryptographic uses, you can use a pseudorandom number generator (PRNG). John von Neumann developed the first one in 1946. His algorithm was not complicated.

> *His idea was to start with an initial random seed value, square it, and slice out the middle digits. If you repeatedly square the result and slice out the middle digits, you'll have a sequence of numbers that exhibit the statistical properties of randomness.*
>
> *"A History of Random Numbers"*[1]

The article quoted above thoroughly explores the evolution of PRNGs and the vulnerabilities you have to look out for when using them. One algorithm that has stood the test of time is the Mersenne Twister[2], covered by Eric Mann in Security Corner: Twist and Shout[3]. While not cryptographically secure, it is widely implemented and useful; given a specific seed, you can generate the same sequence of random values every time.

Cryptographically Secure PRNGs

A Cryptographically secure PRNG meets strong statistical requirements such that it can be used in sensitive applications, like securing Wi-Fi communications or encrypting passwords. For security, the random numbers generated for encryption keys or application nonces should not be guessable by an attacker. For example, if you have a sequence of random numbers from the Mersenne Twister, in theory, you could reconstruct the sequence of numbers that came before your sequence. CSPRNGs must be built to withstand inspection.

Lavarand

Just how critical is having a "true" source of randomness? Cloudflare uses a wall of lava lamps known as LavaRand[4] to generate unpredictable input for the CSPRNGs. It's an extra source of randomness using the unpredictable bubbling and falling "lava" flows.

Recap

Last month's puzzle challenged you to "harness randomness such that we can guarantee that a given input will produce the same random sequence."

> *Given a positive integer, produce a sequence of six colors. The colors can be red, green, blue, yellow, purple, or black. The same integer input must always produce the same sequence of colors in the same order. Varying the input integer by one digit must produce a new sequence that is entirely different such that you can predict how the sequence changes when the integer input changes.*

Given `672985`, your code should consistently produce the same sequence `green-purple-black-purple-black-purple`.

Solutions

For this challenge, you don't need a CSPRNG. In fact, you don't want one at all. PHP's `rand()`[5] and `mt_rand()`[6] functions both use the Mersenne Twister to generate random numbers. Prior to PHP 7.1, `rand()` used a different algorithm, but they are now one and the same. The other key to unlocking a solution is the `mt_srand()`[7] function. It allows us to seed our random number generator. As long as we use the same seed,

1 *"A History of Random Numbers":* https://tashian.com/articles/a-brief-history-of-random-numbers/

2 *the Mersenne Twister:* https://en.wikipedia.org/wiki/Mersenne_Twister

3 *Security Corner: Twist and Shout:* https://www.phparch.com/2019/11/security-corner-twist-and-shout/

4 *LavaRand:* https://blog.cloudflare.com/randomness-101-lavarand-in-production/

5 *rand():* https://php.net/rand

6 *mt_rand():* https://php.net/mt_rand

7 *mt_srand():* https://php.net/mt_srand

Listing 1.

```php
<?php
// set a seed
mt_srand(672985);

$colors = [
    'red', 'green', 'blue', 'yellow', 'purple', 'black'
];

$sequence = [];

for ($i = 0; $i < 6; $i++) {
    $pick = rand(0, 6);
    $sequence[] = $colors[$pick];
}

echo join('-', $sequence);
```

every execution of our code will produce the same "random" set of values. If you don't set the seed, PHP will set it automatically. However, your program will use a different seed each time it runs.

Given these two insights, Listing 1 shows a concise solution. Each time we run it, we'll get the same sequence of colors.

If you prefer a functional approach without loops, check out the code in Listing 2.

Guess the Colors

Listing 2.

```php
<?php
// set a seed
mt_srand(672985);

$colors = [
    'red', 'green', 'blue', 'yellow', 'purple', 'black'
];

$sequence = [];

// a functional approach
$sequence = range(0, 5);
$sequence = array_map(function() use ($colors) {
    $pick = rand(0, 6);
    return $colors[$pick];
}, $sequence);

echo join('-', $sequence);
```

For next month's challenge, we'll build on this month's solution. Write a program that allows users to guess a random sequence of colors. When a user enters a guess, the program should indicate:

1. If a color is 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. OR if the user has guessed the correct sequence.

Dean Hochman from Overland Park, Kansas, U.S., CC BY 2.0[1], via Wikimedia Commons

1 *CC BY 2.0: https://creativecommons.org/licenses/by/2.0*

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 web page or the command line.

Some Guidelines And Tips

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

- *Each solution is encapsulated in a function or a class, and I'll give sample output to test your solution against.*
- *You're encouraged to make your first attempt at solving the problem without using the web for clues.*
- *Refactoring is encouraged.*
- *I'm not looking for speed, cleverness, or elegance in the solutions. I'm looking for solutions that work.*
- *Go ahead and try many solutions if you like.*
- *PHP's interactive shell (`php -a` at the command line) or 3rd party tools like PsySH[8] 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.*

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

8 *PsySH: https://psysh.org*

Survival of the Fiendish

Beth Tucker Long

In ages past, the claim has been that the "fittest" survive. Nowadays, the fittest appear to fall by the wayside, crowded out by those fiendishly over-promising, under-delivering, and making it unbearably difficult to escape.

I spent the greater part of yesterday and today in annoying agony. What had I done to deserve this fate? Well, I tried to pay a company money for a service they sell. I know, I know…how dare I commit such a heinous crime.

It all started with a call about a website that someone couldn't access. Did a little digging, and it was because the website was not using https, and this person's security restrictions only allow them to visit websites that use https. They asked us to add a secure certificate. Sure. Easy peasy. (If only I had known…)

I checked the docs for the hosting company (which shall remain unnamed but is a company with more than enough resources that they should have their business figured out) and saw that they listed a number of different ways to get a secure certificate activated for a website, both automatic and manual installations from various providers, including them. I started with trying a Let's Encrypt certificate because it was more than enough for this website's needs, and free is a price all my clients love. The hosting company offered an automated way to install and renew Let's Encrypt certificates. I followed the instructions to configure everything needed, then discovered that the automated tool didn't have the permission levels needed to run and did not have the options listed in the support docs. Something clearly changed, and the docs were never updated.

I switched over to the directions for manually installing the certificate. The installation was working splendidly until I hit the certificate verification step. Verification required a directory structure that I did not have access to create on the server, so I hit a dead-end there as well. A little disappointed that the Let's Encrypt route didn't work, I acquiesced and signed up for the hosting company's paid secure certificate option. The certificate wasn't free, but considering how much time it would likely take me to troubleshoot the Let's Encrypt installation issues, it would likely be cheaper overall. I completed the purchase and started through the handy automatic process to install the secure certificate. I almost instantly hit a roadblock. Despite no information explaining this limitation in the purchase process nor the support docs, I discovered, thanks to others on the internet, that the automated secure certificate process only works for websites hosted through their special automated hosting service. Custom websites just using the server space were not supported.

I was again disappointed and now a little angry too, but I paid for this certificate, so I need to keep going with it. I switched over to the instructions for manually installing the purchased certificate. These instructions have not been updated in a long time as the terminology used refers to products that the hosting company phased out years ago. Searching through posts on various helpdesk-type websites, I find enough posts that I am able to get through the process of getting the certificate installed. However, the website will still not load via https. I check the logs. No errors. I run the verifications. Everything looks good. I return to searching the online helpdesk sites for answers. I run into a lot of posts from people with the same issue, but no one who ever resolved their issue. Each time, they either said they switched hosting companies, or they had to contact support to get the issue resolved. Switching companies is not an option today, so I go to contact the support team and discover that there is no way to talk to anyone without paying. There is also no information on how much it will cost to talk to someone. At this point, I'm left with a hosting company that wants to charge me for the opportunity to talk to them to see if it is possible to have them fix the broken product that they just sold to me.

This is not an isolated experience. So many companies are profiting off of treating their customer horribly. They make it easy to sign up for a hundred services with the click of a single button but then require you to find and turn off each of those services individually. They sell you convenient products and only tell you later that they only work if you buy their other products as well. They make it easy to migrate to their service but block all access that would make it easy for you to migrate away from their service, including preventing you from downloading backups of your content.

Brick-and-mortar stores would never get away with this. Can you imagine the outrage if a shopping mall locked the doors and refused to let people leave unless they paid an extra fee to take their personal belongings with them when they left? Or if a grocery store hid the exits so you couldn't find them to leave? Or if a department store snuck items into your pockets while you were shopping and then forced you to pay for them when you tried to leave even though you didn't want them or even know you had them?

It's time for tech companies to do some soul-searching. Instead of holding customers hostage to earn a profit, let's conduct ourselves with honor and respect, creating services and customer service experiences that make customers excited to work with us.

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