www.phparch.com

March 2022
Volume 21 - Issue 3

php[architect]

The Magazine For PHP Professionals

World Backup Day

Backups For Beginners

Hack Your Home With a Pi

ALSO INSIDE

The Workshop:
Queus with Horizon

Education Station:
Software History is Licensing

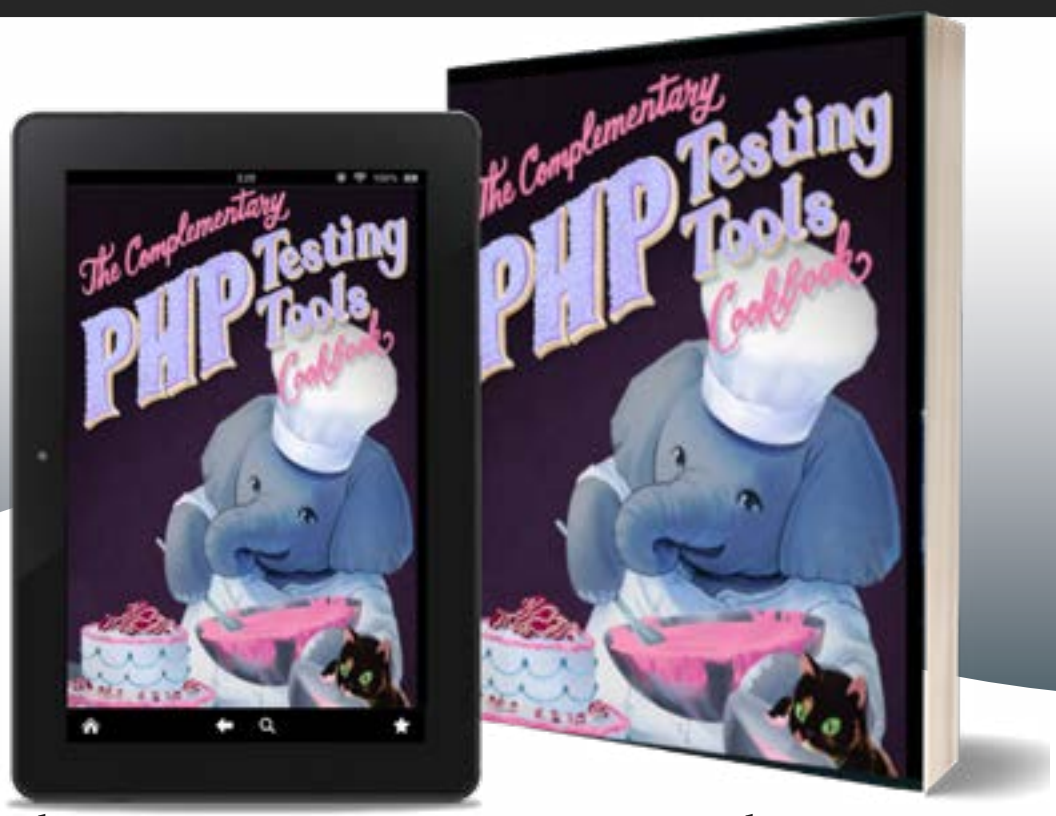
PHP Puzzles:
Finding Prime Factors

Security Corner:
Understanding Supply Chain Security

DDD Alley:
Better Late Than Never

PSR Pickup:
Improving the Developer Experience

finally{}:
I Just Can't



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

MARCH 2022
Volume 21 - Issue 3

php[architect]

Edited with Leprechaun Magic

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

Strengthening Our Weaknesses

John Congdon

As developers, it's very important to identify our weaknesses in order to strengthen our defenses. Learning how to do this earlier on in our careers will help sustain us into the future.

I've been a developer for a very long time. Over 20 years at this point, and there are so many things I wish I would have learned earlier in my career.

Some salt in the wound… as an organizer of San Diego PHP, I've had the opportunity to talk to many talented people over the years. I remember one conversation of me explaining why testing was beneficial.

Me: "Have you ever had that conversation with an employer where you explain that you changed/fixed a piece of code in one feature, and it broke a completely separate feature?"

Them: "No"

Me: a look of absolute disbelief on my face… "Never?"

Them: "No, my code always works."

Whether they were right or not is not the point. There are so many aspects of our career that are in jeopardy every single day if we are not careful. What I learned from small code changes having unintended side effects is that I need to write tests, and to this day, I try to test everything I write to the best of my ability. There are times are a conscious decision is made not to test, but that is happening less and less.

This month's issue helps cover a few more of the potential pain points.

Scott Keck-Warren has a contribution in honor of this month's World Backup Date, *Backups For Beginners*. Follow along for some inspiration in making sure you have a complete backup system. Ken Marks continues his series, *How to Hack Your Home with a Raspberry Pi,* with an article showing how to actually hook up your accelerometer to your Pi and to start storing the data into a database on the Raspberry Pi.

Our columnists also drive home the point of strengthening our weaknesses. In *Understanding Supply Chain Security*, Eric Mann will have you realizing that even our code has a supply chain and that its security is crucial to our success. Joe Ferguson's, *Queues With Horizon* will help us make our application more robust by offloading some processing to give our application the appearance of higher performance. Chris Tankersly brings us into the Education Station with *Software History is Licensing*. Oscar Merida helps us strengthen our abilities while testing out a fun PHP Puzzle, *Finding Prime Factors*. Edward Barnard continues in the DDD Alley with *Better Late Than Never*, where he talks about his own theories around testing. Our newest column by Frank Wallen, PSR Pickup, will start to teach us about the PHP PSR's starting with PSR 0 and 1. And finally{}, brought to you by Beth Tucker-Long is a great piece on burnout titled *I Just Can't*.

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/March2022_code

Backups For Beginners

Scott Keck-Warren

Backups are one of those things we developers don't like to think about until there's a problem, and then it's too late. Anyone who's ever accidentally deleted something only to find out the only good backup is weeks or months old can tell you how important it is that we always have backups as part of our disaster recovery plan. We can't let them slip because not having a functioning backup can cost our companies untold hours and thousands of dollars recreating data. The worst part is apologizing to our users that their data is gone.

In this article, we'll discuss why we should be backing up our data, as well as some options and methods for backing it up.

World Backup Day

You might be wondering why you're reading about backups. It's because World Backup Day is March 31[1]. It's "the day to prevent data loss," according to their site. The idea is to bring awareness of the importance of backups and what can happen if we don't have a good backup of our important data.

It's primarily geared towards consumers, but backups are important for anyone in Development, DevOps, Operations, and IT. If you've been given the keys to a company's critical data, you don't want to be the one to tell the CEO that there's no way to restore data after a ransomware attack or when your largest client's data is gone. Otherwise, you had better be prepared to switch careers.

To that end, I think that every day is World Backup Day for anyone supporting an application, so we should all have it in the back (or front) of our minds.

What is a Backup

To start, let's discuss what a backup is. A backup is defined as a copy of data that is stored elsewhere, so it may be used in the event of data loss to restore the original data. Backups are a form of disaster recovery, and for several types of data backups, can be seen as a way to completely recreate the data. However, some backups may not be able to completely restore all of our data. For example, we may not have a backup of data that is only stored in a caching layer where data is never written to disk. In these cases, we need to plan for these gaps accordingly and be able to act on them.

Why Should I Make A Backup?

As someone who has been working with other people's data for more than 20 years, I've learned the hard way that it's not a question of **if** we'll need our backups but **when**. There are four major categories of data loss we need to be prepared to handle.

Human Error

Human error data loss is due to accidents and unknowingly modifying or deleting data.

This is most likely the one that we'll be responding to the most often as developers. Think of how often someone deleted client A's data when they were attempting to delete client B's data. I have vivid memories of an incident where a support rep (at a previous company) deleted our largest client's data. We had a very angry client calling us as we invented a way to restore just their data on the fly. The CEO wasn't happy with "we're not sure how to get their data back," and it's not something I want to repeat.

File Corruption

File corruption data loss occurs when a software bug causes bad data to be written or when a virus infection corrupts data.

It's the type of data loss that keeps me up at night. Think of all the very public ransomware attacks that have taken down companies, school districts, and even gas pipelines (remember the bags of gas).

Hardware

Hardware failure data loss is caused by drives, RAID controllers, and occasionally CPUs failing.

In a world of virtual data centers, this is becoming less of a worry for our server infrastructure but it's still an issue for company laptops (if you're in charge of that). Think about all the documents sitting on your desktop that you can't recreate. Maybe, that file of passwords you don't have anywhere for security purposes.

Physical Site

The last category of data loss is related to physical destruction. Our data can be lost due to floods, fires, earthquakes, and even theft.

For some real nightmare-inducing results, do a Google Image or YouTube search for "Flooded Server Room." Again due to the move to virtual data centers, this is becoming less and less of a concern.

Thankfully we can reduce these risks by having a sensible backup strategy.

1 *World Backup Day is March 31: http://www.worldbackupday.com*

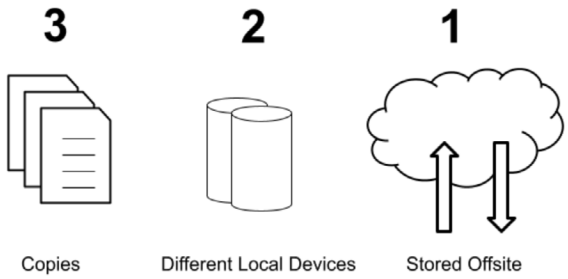
Test Your Backups Regularly

The most important thing about backups is that you test them. Test that they can be read; test that you can install the software to do the restores; and test that the media is still readable. It doesn't do us any good to have a backup if when we need to use them, we can't restore the data, or restoring the data takes too long for it to be useful.

It's best practice to test your backup at least once a year and then whenever there's a significant change to the technology infrastructure. We prefer the idea of testing a "full" backup once a year but then also doing single files at least once a month. We find that November and December tend to be a "quieter" part of the year for our business, so we make that our "let's test our backups time." We also went through a process where we changed Linux distros last year, and we tested our backups before, during, and after the transition.

We also test our disaster recovery plan during the yearly test of our backups. We time ourselves to see how long it takes to go from nothing to a fully running "test" environment. Then we iron out any issues we ran into during the process. I like to "randomly" assign this to someone so anyone who may have to do the full restore can.

If you only take one thing from this article, I hope it's a sticky note to test your backups when you're back at work.



3-2-1 Rule

One of my favorite concepts about how to make sure our data is protected is the 3-2-1 Rule. The Rule can also be referred to as the 3-2-1 Backup Strategy or just 3-2-1 Backups. The basic idea to ensure our data is protected; have 3 copies of our data on 2 different local devices and 1 offsite backup. 3-2-1 allows us to quickly restore from the local backups in cases of Human Error, File Corruption, or Hardware failure and restore if we have a Site related data loss. For our purposes, we will consider them "local backups" if they're in the same physical location.

As an example, I'm writing this article on my MacBook (local copy 1) that's using software to copy my changes to the NAS in my basement (local copy 2) that's automatically uploaded to a cloud service every night (offsite backup 1). I'm not the least bit concerned that I might spill coffee on my laptop, accidentally delete a file, or have my laptop and NAS both stolen (knock on wood) because the hardware is easily replaced and my data is backed up.

As developers, if we're using a distributed version control system (DVCS) like Git, we get a 3-2-1 Backup almost for free if we're using a third-party provider to host our Git repository and push our changes regularly. It's one of the things I love the most about DVCS.

As you're configuring your servers, make sure the 3-2-1 Rule is protecting your data. It's easy to get to 2-2 or 2-1-1, but putting in the extra effort and money into getting yourself to 3-2-1 can make sure you sleep a lot better at night.

Backups For Virtual Servers?

I'm guessing a lot of the people reading this article are wondering what they should be doing if they have servers in the cloud. If you're in this situation, my suggestion is that you still follow the 3-2-1 Backup Rule and for your offsite backup, use a different cloud provider to store the offsite backup. Cloud storage providers are popping up all over the place, so it's an embarrassment of riches. For the second local copy of your data, I recommend either spinning up a dedicated backup server in a data center or adding an extra virtual drive (on a separate data store) to store the backups if your data is still small enough. If a dedicated backup server is not in your primary data center, you get bonus points.

Talk to your hosting provider and see what options they provide and how granular they can restore data. I've worked with providers that will only do a full VM restore from a once-a-day backup and others that can do from a full VM restore all the way down individual files with hour-level granularity.

I was working on a project a couple of years ago that was using a "full service" hosting provider. They maintained five-nines (99.999%) of availability on the host hardware, VMs, Internet connection, maintained the backups, and even made sure the hosts were up to date on their patches. It was perfect for a "small" project that we weren't planning on doing a lot with after it was in maintenance mode.

Our backup plan came crashing down when the hosting provider encountered a bug in their routers that took their whole data center offline for a full working day. The client was losing money while they were down, and the provider didn't have time to restore the data. We couldn't set up their data in a different location because it was "locked" into that provider's cloud.

If we had set up an "offsite" backup, we could have spun up another server at a different data center.

Full, Incremental, or Differential Backups?

We will discuss four types of backups in this article.

A **full backup** contains all of our data. We'll talk about the pros and cons of this in the "Full Backups Every Day" section below.

A **differential backup** contains all of the changes since the last full backup. In this mode, we make a full backup on a fixed schedule, and then every backup after that contains just the changes. This type of backup makes it easy to restore our data completely; we only have to restore data from the last full backup and then the last differential backup. The downside is that these backups can get quite large if there are a lot of changes to our data.

An **incremental backup** contains all of the changes from the previous backup. The important difference between an incremental backup and a differential backup is that incremental backups use the previous backup of any type and not the previous full backup. This type of backup will likely cause each backup to be smaller. For a full restore, you will have to restore the last full backup and then each incremental backup which could be many backups depending on your schedule.

The previous backup methods have existed for a long time, but the most recently added backup method is **continuous data protection (CDP)**. CDP is different from a traditional backup in that we constantly backup up our data, so we don't need to specify a time to do backups, and we can restore to any point in time. There is additional overhead to this process because every time we perform a write or delete, we have to perform it twice (or three times). If our backups are on a separate server (which they should be), we have to perform some network communication. If we're developing software that needs to have this level of data retention, it can be built into the application logic, and some backup software supports it as well.

Retention Time

Most backups specify how long they should be kept; also known as the retention time, and it's an important part of the backup lifecycle. We could keep a record of every version of every file we ever write, but it can get very expensive buying more and more storage for our backups. To this end, we define the amount of time we keep each backup, and once this time has elapsed, we can reuse the media that the "old" backup was using.

This period is going to depend on a lot of factors. If we have regulatory or contractual requirements to keep a backup of data for a specific period, this will be the most important. Next is budgetary because we're going to have to justify why we need to keep backups for as long as we do. Finally, there is usually a discussion to determine if a backup from a year ago will be useful for your business.

A good rule of thumb is to try and keep 3 months' worth of backups. Anecdotally, data loss tends to be found close to when it happens or after a very long time (and then without being able to remember the name of the deleted file), so having backups for more than that might be unhelpful.

How To Generate Our Backups

Now that we've discussed, at great length, a lot of underlying backup terminology, let's discuss some backup options. These options determine what type of backup we'll make each day, how long we'll keep each backup, and where the backups will be stored.

Continuous Data Protection

In an ideal world, this is the best possible backup option. We would be able to restore any data at any point in time. However, we don't live in an ideal world, so not every application will support this, and it can have some serious performance issues.

As developers, we can create logic in our applications that will save data to multiple locations to have this level of protection on at least some of our data. For example, we accept data from our clients that they might not be able to recreate (multimedia is the best example). Because we know that data loss isn't an option, we can upload the data into two different buckets at one provider and one at another as soon as the photo is uploaded.

I'm a big fan of using solutions like this on staff machines because then all their working files are backed up regularly.

Full Backups Every Day

The next option is to perform full backups every day. The benefit to this method is that restoring a full backup is as easy as a single restore. The downside to this process is that because we're making a full backup of all our data, the total amount of backup space we have to have available can be extensive and expensive. It will also put more of a CPU, IO, and network load on our infrastructure because we need to copy all our data off the system to our backup server.

That being said, this isn't cost-prohibitive for a lot of applications. Depending on your dataset, it's possible to have full backups of all of your data with several months of retention. We can make retention decisions to reduce the total cost by keeping backups on Sundays for 3 months but only keeping Monday through Saturday backups for two weeks. Then we can still go back 3 months but not necessarily to every day of that period.

Do the math yourself to see what it will cost you. You might find it to be surprisingly affordable.

Full Backups With Incremental or Differential Backups

In the next option, we'll perform a full backup on a schedule (once a month or week) and then incremental or differential backups the rest of the time. Doing so reduces the cost and resource usage because of the reduced amount of data storage and transfer. The downside is that full restores will take longer because we have to restore at least two backups (unless we're lucky enough to only need to use that full backup).

An option for how to schedule this is a Grandfather-father-son scheme.

The grandfather-father-son has three components:

1. Grandfather: A full backup, done once a month, stored separately from the other backups. Ideally, it's offsite, but it should be on a different type of media or a different server.
2. Father: A full backup, done once a week. It could be kept on-site but it would ideally have an offsite copy.
3. Son: An Incremental or differential backup to the same storage as the father.

Test Your Backups Today

In this article, we discussed that it's important to make sure we have backups because it's not a question of if we lose data but when. We need to make sure we follow the 3-2-1 backup method and regularly test our backups. We also discussed different backup methods like CDP, Full Backups every day, and Grandfather-father-son.

If you've made it this far and haven't written "test backups" on a sticky note, please do it now, or you can't say I didn't warn you.

Scott Keck-Warren is the Directory of Technology at WeCare Connect and has been working professionally as a PHP developer for over a decade, as well as leading technical teams. Scott creates content about topics of interested for PHP developers in prose at thisprogrammingthing.com and in videos at http://www.phpdevelopers.tv/. @scottkeckwarren

How to Hack your Home with a Raspberry Pi - Part 3 - Storing Sensor Data on your Raspberry Pi

Ken Marks

Welcome back to another installment of 'How to Hack your Home with a Raspberry Pi.' At the end of this article, you should have a Raspberry Pi with an accelerometer sensor connected to the General Purpose Input Output (GPIO) bus that automatically stores sensor data to a database. So grab your Raspberry Pi and settle in so you can continue this journey with me as we connect the accelerometer sensor and create a Unix Service!

Figure 3.

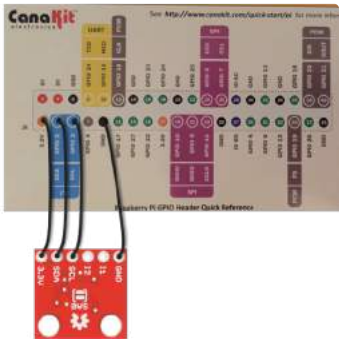
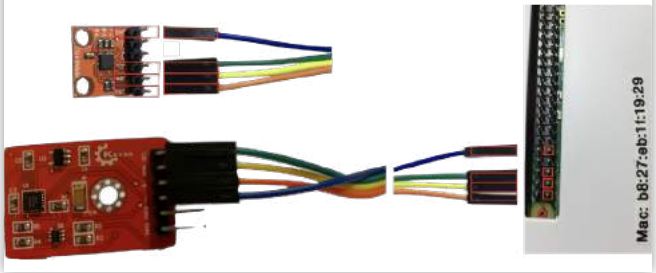


Figure 4.



Introduction

In part 1, I covered:

- The story behind my motivation for this project
- A list of components needed (including the Raspberry Pi)
- How to install the Operating System on the Pi
- Configuring the Pi

Last month (in part 2), I covered:

- Updating the software packages on your Pi
- Installing and testing the LAMP Stack

So please make sure you start from the beginning of the series if you want to build this project yourself.

In this installment, we will:

- connect up an accelerometer sensor
- create a database to store the accelerometer data
- compile a C/C++ program that reads the accelerometer data from the sensor and logs it to the database
- and create a Unix Service that:
  - automatically starts the data logging when the Pi boots
  - and stops the data logging when we shut our Pi down

Figure 1.

Figure 2.

Connecting Sensors

In part 1 of this series, I specified two different types of accelerometers you can use, along with a ribbon cable with female pins on each end. I'm using the Keyes MMA8452Q from eBay[1], and I've peeled off 4 wires from the female to female 40 pin ribbon cable I purchased from Amazon[2]:

It is important to correctly wire the accelerometer to the Raspberry Pi. Figure 3 shows you how to hook up the accelerometer to the GPIO bus of the Raspberry Pi.

Figure 4 shows you the slight differences in the connections between the two accelerometers I've specified.

The four connections are:

Pin Name	Accelerometer Connection	Raspberry Pi GPIO Connection
Power	3.3V or +	3.3V (pin 1)
Data signal	SDA	SDA [GPIO2] (pin 3)
Clock signal	SCL	SCL [GPIO3] (pin 5)
Ground	GND or -	GND (pin 9)

1 *eBay: https://phpa.me/ebay-accelerometer*

2 *Amazon: https://phpa.me/amazon-fem-jumper-40*

Carefully connect your accelerometer to your Raspberry Pi with the four-pin ribbon cable and double-check your connections are correct before powering up the Pi.

Various data communication protocols are used to connect sensors to microcontrollers and the Raspberry Pi supports several of them. One popular protocol is called Inter-Integrated Circuit (I2C) Bus[3]. The I2C bus is a serial communication (2 wire) bus that allows you to connect multiple low-powered devices to a computer or microcontroller using a 2-byte address to reach each sensor on the bus.

Power Up Your Pi and Remotely Log in

After your accelerometer is connected to your Pi, 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 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 always like to keep my Pi's OS up-to-date, so I'll first check to see if any software packages need upgrading using the apt package manager and run the following two commands in the terminal window:

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

Logging Data

Now we are ready to configure our Pi to start collecting data. But first, we need to create a database to store the sensor data.

Create a Database for Storing Accelerometer Data

Our accelerometer reports sensor data in three-axis (X, Y, and Z). We'll create a database called `AccelerometerData` with one table called `accelerometer_data`, which will store a time-stamp along with x, y, and z-axis accelerometer data in units of g-force.

In the terminal window, log into the MySQL CLI as super-ruser by typing the following command:

```
sudo mysql
```

You are now in the MariaDB (MySQL) CLI tool.

3 *Inter-Integrated Circuit (I2C) Bus:* https://en.wikipedia.org/wiki/I%C2%B2C

Create a Database User for Our Database

Let's create a database user called `accelerometer` with a password of `accelerometer` that will have privileges granted on all tables in the `AccelerometerData` database by typing the following commands:

```
CREATE USER 'accelerometer'@'localhost'
    IDENTIFIED BY 'accelerometer';
GRANT ALL PRIVILEGES ON AccelerometerData.*
    TO 'accelerometer'@'localhost';
FLUSH PRIVILEGES;
```

After executing these commands successfully, you should see `Query OK, 0 rows affected` in the MySQL CLI.

Type `exit` to exit out of the CLI in the terminal window and return you to the shell.

Create the Database

Now we can create the `AccelerometerData` database. We will use the web-based database management tool Adminer we installed in the last installment of this series (part2).

In your browser on your computer, bring up a tab and navigate to `http://raspberrypi.local/adminer`. Login as `accelerometer` with a password of `accelerometer` into the Adminer tool.

Select the Create database link. Create the `AccelerometerData` database and select the Save button. See Figure 5. Adminer will show that the `AccelerometerData` database has been created.

Figure 5. Adminer: Save AccelerometerData database

Download Listing 1, the `accelerometer_data.sql` SQL file to your computer.

Typically, database tables use the InnoDB engine, which stores them on a storage partition attached to the OS. In

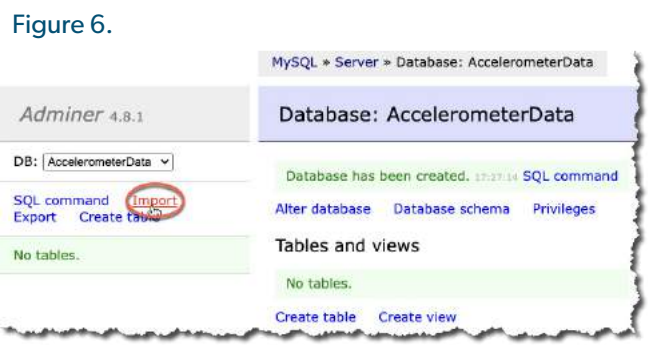
Listing 1.

```
1.  SET NAMES utf8;
2.  SET time_zone = '+00:00';
3.  SET foreign_key_checks = 0;
4.  SET sql_mode = 'NO_AUTO_VALUE_ON_ZERO';
5.
6.  DROP TABLE IF EXISTS `accelerometer_data`;
7.  CREATE TABLE `accelerometer_data` (
8.    `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
9.    `created` datetime(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),
10.   `axis_x` float NOT NULL,
11.   `axis_y` float NOT NULL,
12.   `axis_z` float NOT NULL,
13.   PRIMARY KEY (`id`)
14. ) ENGINE=MEMORY DEFAULT CHARSET=latin1;
```

our case, that is the SD card of the Raspberry Pi. Doing so is acceptable for normal use where we create, read, and write to files. However, we will be writing accelerometer data to our database ten times a second, which will cause the SD card to fail eventually. Since we only plan to keep the last minute of data (and delete the older data), storing our table data in RAM (MEMORY) will be more appropriate and help with the longevity of our SD card.

In Adminer, import the `accelerometer_data.sql` file into the newly created `AccelerometerData` database. Select the Import link shown in Figure 6.

Figure 6.



Select the Choose Files button to select the `accelerometer_data.sql` file and select the Execute button to import the SQL file. Adminer will show the SQL data has been imported into the `AccelerometerData` database and created a table called `accelerometer_data`.

Take a look at the `accelerometer_data` table by clicking on the accelerometer_data link. You will see its description as shown in Figure 7.

The `id` field is set as a `bigint` to simplify any issues with a potential rollover of primary key values. This way, if the Pi were plugged in and ran without any loss of power, reset, or another failure, it would take almost 585 billion years before the primary key rolled over.

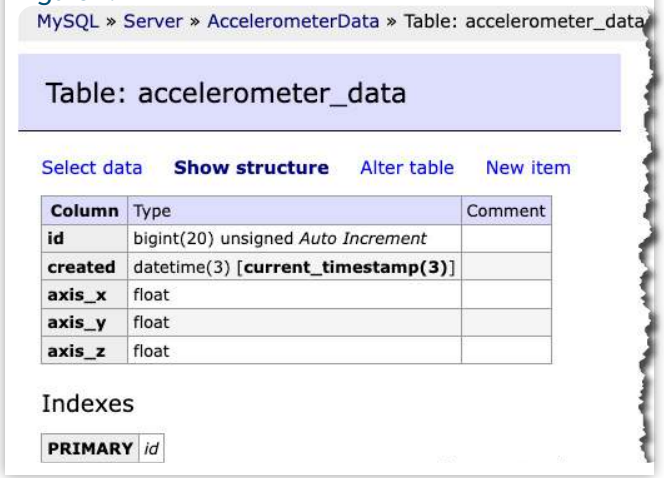
We will be storing a reading from the accelerometer every 100 milliseconds. Therefore, we need to have a timestamp capable of storing fractional seconds. The created field is a `datetime(3)` to capture the number of milliseconds in the time-stamp. It also has a default value of `[current_timestamp(3)]` stored ON UPDATE.

The three fields: axis_x, axis_y, and axis_z, are set to float to store a gravity value that will normally be between 0 and 1 when the accelerometer is not being shaken in the axis that is perpendicular to the ground.

Log the Accelerometer Data

I have created a C++ program to log 60 seconds of accelerometer data every 100 milliseconds. It will store 600 rows (maximum) of data, deleting the oldest rows. In addition to this program, we need to install three software packages: two to communicate with the MySQL database and another to communicate with the accelerometer sensor.

Figure 7.



Install a C++ Mysql Connector

For the C++ program to interface with the database, we need to install the MySQL database development libraries and C++ connector. Doing so will also install the development tools we need to compile our C++ code into an executable program. In the terminal window, enter the following command:

```
sudo apt install default-libmysqlclient-dev \
libmysqlcppconn-dev
```

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. The installation could take a couple of minutes because there are a number of dependencies needed for these packages to work.

Install the I2c Tools

Next, we need to make sure our program can talk to the accelerometer sensor we just connected. Our accelerometer is connected to the I2C bus on the GPIO connector of the Pi, and in part 1 of this series, we enabled the I2C interface to allow communication to I2C devices connected. For our program to communicate with the accelerometer sensor, we need to install some I2C drivers. In the terminal window, enter the following command:

```
sudo apt install i2c-tools
```

When 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.

Download the Accelerometer Data Logging Program

Since we're going to do some compiling on the Pi, let's create a subdirectory in our home folder called `accelerometer`. In the terminal window, enter the following commands:

```
cd
mkdir accelerometer
cd accelerometer
```

In the terminal window, download the `log_accelerometer_data.cpp`[4] C++ file to your Pi using `wget`:

```
wget -O log_accelerometer_data.cpp https://phpa.me/pi-accel
```

Before we compile the code, lets take a closer look and see how it works. Often in C/C++ code, there is some boiler-plate code we need to setup before we can get to the actual work we need to do. First, we need to include a bunch of library function definitions. See Listing 2.

Listing 2.

```
1. #include <iostream>
2. #include <unistd.h>
3. #include <string>
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include <linux/i2c-dev.h>
7. #include <sys/ioctl.h>
8. #include <fcntl.h>
9. #include "/usr/include/mysql/mysql.h"
10.
11. using namespace std;
```

Many of these include files are so we can use the standard C/C++ library functions, output to the terminal window, and use file I/O. The other includes are for communicating with the I2C bus and the MySQL library. I am also specifying we are using the `std` namespace, so all we have to do is preface all our calls to the standard library with `std::`

Next, we have some references for communicating with the MySQL library:

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

And some definitions for our credentials to connect to the database:

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

Then there is a definition for clearing the terminal window screen for displaying accelerometer data during testing:

```
// Cursor Movement: Clear screen to print
// readable output of axis data
#define clearScreen() printf("\e[1;1H\e[2J")
```

Next, we have a function that will read the data from all three axes of the accelerometer sensor using the passed in file descriptor, convert the data into g-force, then assign the results to the passed in axis parameters. A fantastic resource for how this accelerometer works and how to use it is over at Sparkfun[5]. They have links to the datasheet, a tutorial on how to hook it up to an Arduino, and give you some code to use to communicate with it.

Most I2C devices have a 7-bit address sent in a single hexadecimal byte. Many devices also have registers we need to write to configure the device for certain functions we want it to perform. To spare you the pain of having to wade through the datasheet on your own, I'll break down the code as we go starting with Listing 3.

Almost everything we communicate with in a Unix-based OS is treated as a file and requires a file descriptor. The first thing we need to do is use the `ioctl()` function to connect the file descriptor that references the I2C bus to our accelerometer, located at hexadecimal address `0x1D`.

Next, we'll create an array of two `chars` (`config[2]`). The first byte is the register we want to write to, and the second byte is the value we want to write to this register. Writing the value `0x00` to the mode register (`0x2A`) puts the accelerometer into Standby mode. Next, we `write()` to our file descriptor the value `0x01` to the mode register to put the accelerometer into Active mode. Doing so will clear out all the registers on the device. Then we need to set up the accelerometer to read

Listing 3.

```
1. void getAccelerometerData(
2.                           int& fileDescriptor,
3.                           float& xAxis,
4.                           float& yAxis,
5.                           float& zAxis
6.                           ) {
7.     // Get I2C device, MMA8452Q I2C address is 0x1D(29)
8.     ioctl(fileDescriptor, I2C_SLAVE, 0x1D);
9.
10.    // Select mode register(0x2A)
11.    // Standby mode(0x00)
12.    char config[2] = {0};
13.    config[0] = 0x2A;
14.    config[1] = 0x00;
15.
16.    write(fileDescriptor, config, 2);
17.
18.    // Select mode register(0x2A)
19.    // Active mode(0x01)
20.    config[0] = 0x2A;
21.    config[1] = 0x01;
22.    write(fileDescriptor, config, 2);
23.
24.    // Select XYZ data configuration register(0x0E)
25.    // Set range to +/- 2g(0x00)
26.    config[0] = 0x0E;
27.    config[1] = 0x00;
28.    write(fileDescriptor, config, 2);
29.
30.    usleep(5000); // 5 ms
31.        ...
```

4 `log_accelerometer_data.cpp`: https://www.phparch.com/downloads/log_accelerometer_data.cpp

5 *Sparkfun:* https://phpa.me/sparkfun-accelerometer

g-force values between -2 and +2. We do this by writing the value `0x00` to the XYZ data configuration **register (`0x0E`). We then sleep for 5 milliseconds to allow the accelerometer enough time to stabilize before taking a reading.

Now we'll take a look at the rest of the function in Listing 4, which reads the accelerometer and converts the readings to g-force.

Now that we've configured the accelerometer, we can read the g-force data for all three axes. We do this by creating an array of 7 `chars` (`data[7]`), then we `read()` from our file descriptor into this array. If there is not an I/O error, we can process the data we get back from the accelerometer.

The data in element 0 contains status information on each axis. The actual acceleration data is contained in elements 1-7, two bytes per axis. This accelerometer reports data for each axis with 12 bits of accuracy expressed in 2's complement numbers. The most significant byte (MSB) for the X-axis is contained in element 1 and the least significant byte (LSB) is contained in element 2. To convert this data into a 12-bit binary number, we use the following code:

```
// Convert the data to 12-bits
int xAccl = ((data[1] * 256) + data[2]) / 16;
if(xAccl > 2047)
{
        xAccl -= 4096;
}
```

Now our X-axis data is contained in the integer `xAccl`. Then we need to scale this integer to the input range and save it to the floating-point parameter `xAxis` passed into the function using the following code:

```
int scale = 2;

xAxis = (float) xAccl / (float)(1<<11) * (float)(scale);
```

All of this bit manipulation is known as bit twiddling[6].

Finally, we can output the accelerometer data for each axis in the terminal window (stdout) with this code:

```
// Display to stdout
clearScreen();
printf("G-Force in X-Axis : %f \n", xAxis);
printf("G-Force in Y-Axis : %f \n", yAxis);
printf("G-Force in Z-Axis : %f \n", zAxis);
```

Now let's take a look at the *main* routine that calls our `getAccelerometerData()` function and inserts the data into the `accelerometer_data` table of our `AccelerometerData` database.

First, in Listing 5, we will open a connection to the I2C bus or exit out if we fail.

6 *bit twiddling:* https://graphics.stanford.edu/~seander/bithacks.html

Listing 4.

```
1. ...
2. // Read 7 bytes of data
3. // status, xAccl msb, xAccl lsb, yAccl msb,
4. //   yAccl lsb, zAccl msb, zAccl lsb
5. char data[7] = {0};
6.
7. if(read(fileDescriptor, data, 7) != 7)
8. {
9.     printf("Error : Input/Output error \n");
10. }
11. else
12. {
13.     // Convert the data to 12-bits
14.     int xAccl = ((data[1] * 256) + data[2]) / 16;
15.     if(xAccl > 2047)
16.     {
17.         xAccl -= 4096;
18.     }
19.
20.     int yAccl = ((data[3] * 256) + data[4]) / 16;
21.     if(yAccl > 2047)
22.     {
23.         yAccl -= 4096;
24.     }
25.
26.     int zAccl = ((data[5] * 256) + data[6]) / 16;
27.     if(zAccl > 2047)
28.     {
29.         zAccl -= 4096;
30.     }
31.
32.     int scale = 2;
33.
34.    xAxis = (float) xAccl / (float)(1<<11) * (float)(scale);
35.    yAxis = (float) yAccl / (float)(1<<11) * (float)(scale);
36.    zAxis = (float) zAccl / (float)(1<<11) * (float)(scale);
37.
38.     // Display to stdout
39.     clearScreen();
40.     printf("G-Force in X-Axis : %f \n", xAxis);
41.     printf("G-Force in Y-Axis : %f \n", yAxis);
42.     printf("G-Force in Z-Axis : %f \n", zAxis);
43. }
```

Listing 5.

```
1. int main()
2. {
3.     // Open connection to I2C bus
4.     int fileDescriptor;
5.     char bus[] = "/dev/i2c-1";
6.
7.     if((fileDescriptor = open(bus, O_RDWR)) < 0)
8.     {
9.         printf("Failed to open the bus. \n");
10.        exit(1);
11.    }
12.        ...
```

Listing 6.

```
1. //initialize database connection
2. mysql_init(&mysql);
3.
4. // the three zeros are:
5. // Which port to connect to, which socket to connect to
6. // and what client flags to use.  unless you're changing
7. // the defaults you only need to put 0 here
8. connection = mysql_real_connect(
9.         &mysql, HOST, USER, PASSWD, DB, 0, 0, 0);
10.
11. // Report error if failed to connect to database
12. if (connection == NULL)
13. {
14.     cout << mysql_error(&mysql) << endl;
15.     return 1;
16. }
```

Listing 8.

```
1. while(1)
2. {
3.         ...
4.         if (num_of_rows >= 600)
5.         {
6.             string oldest_entry_query =
7.                 "SELECT id FROM accelerometer_data "
8.                 + "ORDER BY created ASC LIMIT 1";
9.
10.            // Send oldest entry query to database
11.            query_state = mysql_query(
12.            connection,
13.            oldest_entry_to_delete.c_str());
14.
15.            if(query_state != 0)
16.            {
17.                cout << mysql_error(connection) << endl;
18.                return 1;
19.            }
20.
21.            result = mysql_store_result(connection);
22.
23.            string id_to_delete = mysql_fetch_row(result)[0];
24.
25.            string oldest_entry_to_delete =
26.            "DELETE FROM accelerometer_data WHERE id = "
27.            + id_to_delete;
28.
29.            // Send delete query to database
30.            query_state = mysql_query(
31.            connection,
32.            oldest_entry_to_delete.c_str());
33.
34.            if(query_state != 0)
35.            {
36.                cout << mysql_error(connection) << endl;
37.                return 1;
38.            }
39.
40.            // Free result or a memory leak will occur
41.            mysql_free_result(result);
42.        } // End IF 600 or more rows
43.        ...
44. } // End Forever Loop
```

Listing 7.

```
1. ...
2. // Forever Loop
3. while(1)
4. {
5.     string accelerometer_count_query =
6.         "SELECT COUNT(*) FROM accelerometer_data";
7.
8.     // Send count query to database
9.     query_state = mysql_query(
10.        connection,
11.        accelerometer_count_query.c_str());
12.
13.    if(query_state != 0)
14.    {
15.        cout << mysql_error(connection) << endl;
16.        return 1;
17.    }
18.
19.    // store result
20.    result = mysql_store_result(connection);
21.
22.    int num_of_rows = atoi(mysql_fetch_row(result)[0]);
23. ...
```

In Listing 6, we will initialize and open our database connection or report an error and exit out if we fail.

Next, we have an infinite loop that will do the following work:

- If there are 600 or more rows, delete the oldest row
- Call the `getAccelerometerData()` function to get the current accelerometer data
- Insert the accelerometer data into the database

The loop effectively stores the latest 60 seconds worth of data, ten measurements per second.

Let's take a look at this code in Listing 7. Within our infinite loop, we'll start by getting the number of rows in the `accelerometer_data` table and convert the result to an integer so we can use it in a conditional statement.

In Listing 8, we'll check to see if there are 600 or more rows in the table and delete the oldest one if there is.

If there are 600 or more rows, we delete the oldest row by querying for the `id` field in ascending order, ordered by the `created` field, and then delete it.

As you can see, we need to make quite a number of calls to set up a query to the database and get the results in C/C++. We also have to remember to free the contents of the `result` pointer variable, or we will have a memory leak. It makes me appreciate the simplicity of using PHP to communicate with the database. Another thing to note is this code only deletes the oldest row if there are 600 or more rows. A more robust solution would delete ALL rows older than 60 seconds. To implement that would add a little more complexity and obscure the main point of what we are doing here. So, I leave the performance improvements to you as an exercise. In practice, this code works well because the timing is very tight. As long as you are not manipulating the database outside of this program, this should be good for prototypical experimentation.

Let's continue by looking at the rest of the code after the condition block. See Listing 9. This code first calls the `getAccelerometerData()` passing in three floating-point values (by reference) that we created to store the three axes of accelerometer data, along with the file descriptor referring to our I2C bus. Next, we insert the accelerometer data into the `accelerometer_data` table. And finally, we put the program to sleep for 100 milliseconds. Once the program wakes up, the loop repeats. Forever.

The complete listing[7] for `log_accelerometer_data.cpp` is available for download.

Build the Accelerometer Data Logging Program

Now that you understand how this program works, let's compile it.

When we installed the `default-libmysqlclient-dev` and `libmysqlcppconn-dev` packages, the `g++` GNU C/C++ compilation tools were also installed as dependencies. We need to compile the code in the log_accelerometer_data.cpp file directly on the Pi itself. Doing so will ensure the executable generated is code that the Pi can understand. In the terminal window, execute the following command:

```
g++ -std=c++11 -o log_accelerometer_data \
log_accelerometer_data.cpp -lmysqlclient
```

This command runs the `g++` compiler, specifying the standard **C++ 2011 library. `log_accelerometer_data` is the name of the output executable file, `log_accelerometer_data.cpp` is the name of the source code file, and the `-lmysqlclient` specifies this program needs to use the MySQL client library. You should not see any output or errors in the terminal window. After building and executing a long listing, the terminal window should look like Figure 8.

Figure 8.

```
pi@raspberrypi:~/accelerometer $ g++ -std=c++11 -o log_accelerometer_data log_ac
celerometer_data.cpp -lmysqlclient
pi@raspberrypi:~/accelerometer $ ls -al
total 44
drwxr-xr-x 2 pi pi  4096 Feb 16 13:24 .
drwxr-xr-x 6 pi pi  4096 Feb 15 17:13 ..
-rwxr-xr-x 1 pi pi 27924 Feb 16 13:24 log_accelerometer_data
-rw-r--r-- 1 pi pi  5314 Feb 16 12:57 log_accelerometer_data.cpp
pi@raspberrypi:~/accelerometer $
```

As you can see, we now have an executable program that we can run to get acceleration data and store it in the database. Let's run it and take a look at the output. Type the following in the terminal window to run the program:

```
./log_accelerometer_data
```

If you orient the accelerometer sensor flat on a table with the components facing up, the Z-axis will be perpendicular to the ground, as shown in Figure 9.

Listing 9.

```
1.  ...
2.                  } // End IF 600 or more rows
3.
4.          float x;
5.          float y;
6.          float z;
7.
8.          getAccelerometerData(fileDescriptor, x, y, z);
9.
10.         string accelerometer_insert_query =
11.             "INSERT INTO accelerometer_data "
12.             + "(axis_x, axis_y, axis_z) VALUES ("
13.             + std::to_string(x) + ", " + std::to_string(y)
14.             + ", " + std::to_string(z) + ")";
15.
16.         // Send insert query to database
17.         query_state = mysql_query(
18.                     connection,
19.                     accelerometer_insert_query.c_str()
20.                     );
21.
22.         if(query_state != 0)
23.         {
24.             cout << mysql_error(connection) << endl;
25.             return 1;
26.         }
27.
28.         usleep(100000); // 100 milliseconds
29.     } // End Forever Loop
30.
31.     mysql_close(&mysql);
32.
33.     return 0;
```

Figure 9.

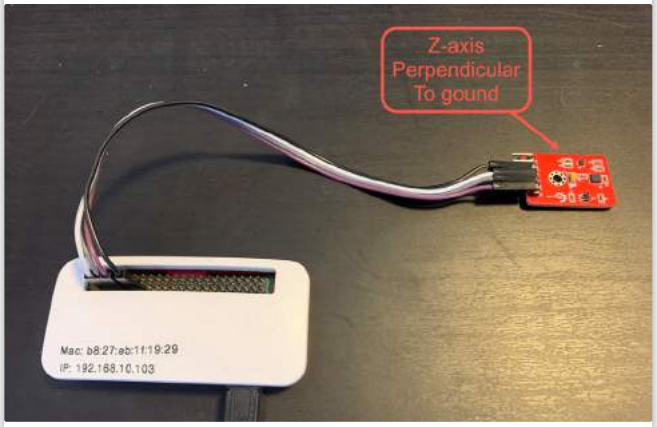


Figure 10.

7 *listing:* *https://www.phparch.com/downloads/log_accelerometer_data.cpp*

Figure 11.

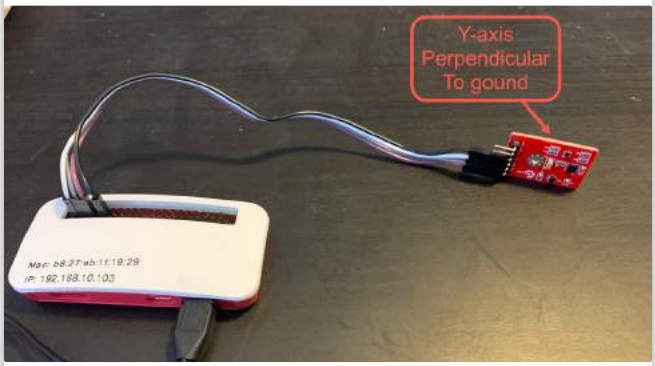
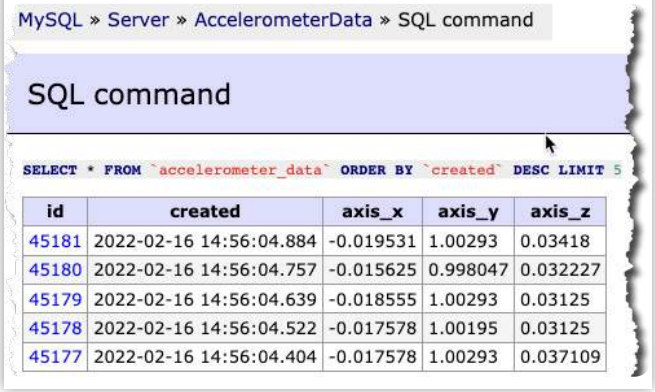


Figure 12.

```
G-Force in X-Axis : -0.020508
G-Force in Y-Axis : 1.003906
G-Force in Z-Axis : 0.018555
```

Figure 13.

Figure 14.



id	created	axis_x	axis_y	axis_z
45181	2022-02-16 14:56:04.884	-0.019531	1.00293	0.03418
45180	2022-02-16 14:56:04.757	-0.015625	0.998047	0.032227
45179	2022-02-16 14:56:04.639	-0.018555	1.00293	0.03125
45178	2022-02-16 14:56:04.522	-0.017578	1.00195	0.03125
45177	2022-02-16 14:56:04.404	-0.017578	1.00293	0.037109

Listing 10.

```
1. [Unit]
2. Description=Accelerometer Data Logging Service
3. After=multi-user.target
4.
5. [Service]
6. ExecStart=/root/log_accelerometer_data
7. StandardOutput=null
8.
9. [Install]
10. WantedBy=multi-user.target
11. Alias=accelerometerdatalogging.service
```

You should see the output in Figure 10 in your terminal window (almost 1 gravity unit of force on the Z-axis).

Now, If you grab the wires going to the accelerometer sensor and rotate it 90 degrees to the right (Figure 11), the Y-axis will be perpendicular to the ground. Then, you should see the output shown in Figure 12.

Once the program has been running for a minute, we will only be keeping the latest 60 seconds of data, so let's take a look at the database.

Use the browser on your computer to log in as `accelerometer` with a password of `accelerometer` in the Adminer tab (`http://raspberrypi.local/adminer`).

Select the AccelerometerData database and then select the accelerometer_data table. See Figure 13. You will notice there are always 600 rows stored every time you select all the data from the accelerometer_data table.

In Adminer, let's execute the following SQL command:

```
SELECT * FROM `accelerometer_data`
ORDER BY `created` DESC LIMIT 5
```

We can see the 5 latest readings stored from the accelerometer in Figure 14. If you execute the query again, you will see newer data stored from the accelerometer. We know our program is working correctly. Exit out of our program by typing ctl+ C in the terminal window.

Creating and Installing a Unix Service

To avoid the pain of manually starting the `log_accelerometer_data` program every time we want to use it, we will create a Unix service that loads the program when the Raspberry Pi boots and unloads it when the Pi shuts down. Here is a link that explains how to create a Unix service[8] that runs a python script.

Create the Unix Service Configuration File

We need to create a Unix service Unit configuration file. In the terminal window (in the `~/accelerometer`folder), open up the nano editor to create a file called `accelerometerdatalogging.service` using the following command:

```
nano accelerometerdatalogging.service
```

Enter the the code in Listing 10, then type `ctl+O` to write the file and `ctl+X` to exit nano.

The systemd init system manages when services are run, and devices are mounted to the OS. The `[Unit]` section specifies the name of the service with the `Description=` directive and which units are started before starting this unit using the `After=` directive. The `[Service]` section specifies where the program can be found with the `ExexStart=` directive (usually in the root folder), and whether we want output to the

8 *Unix service:* https://www.raspberrypi-spy.co.uk/?p=5394

Figure 15.

```
pi@raspberrypi:~ $ sudo systemctl enable accelerometerdatalogging.service
Created symlink /etc/systemd/system/accelerometerdatalogging.service → /lib/syst
emd/system/accelerometerdatalogging.service.
Created symlink /etc/systemd/system/multi-user.target.wants/accelerometerdatalog
ging.service → /lib/systemd/system/accelerometerdatalogging.service.
pi@raspberrypi:~ $
```

terminal or not with the `StandardOutput=` directive—we'll set this to `null` because we don't need the output. The `[Install]` section defines the behavior of our unit when it is started and stopped and by whom using the `WantedBy=` directive. We specify an alias using the `Alias=` directive allowing us to use a name to enable, disable, start, and stop our service. Here is a link for more information on the anatomy of Unix system Unit configuration files[9].

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 log_accelerometer_data /root
sudo cp accelerometerdatalogging.service \
/lib/systemd/system
```

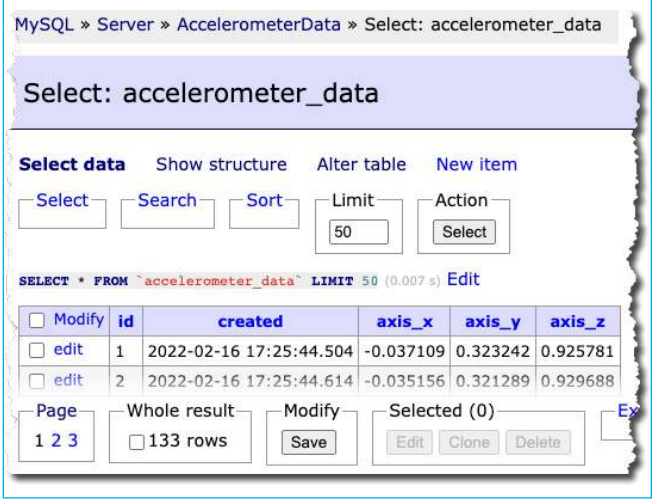
To enable our service using `systemctl`, type the following in the terminal window:

```
sudo systemctl enable accelerometerdatalogging.service
```

Doing so will create a symbolic link that systemd will use to start and stop our accelerometer data logging service. You should see Figure 15 displayed in the terminal window.

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

Figure 16.




```
sudo systemctl start accelerometerdatalogging.service
```

We know the service is running if we go back into Adminer and run a SELECT query on the accelerometer_data table in the AcceleromterData database. See Figure 16.

Finally, let's test whether the service starts back up on a reboot. In the terminal window, by typing the following to reboot the Pi:

```
sudo reboot
```

The reboot will terminate your shell session on the Pi and return you to your shell session on your local computer.

Wait until your Pi fully reboots (30 seconds to a minute), and in Adminer, let's execute the following SQL command a couple of times to verify the service is running correctly:

```
SELECT * FROM `accelerometer_data`
ORDER BY `created` DESC LIMIT 5
```

We are all set up to use this data to create a web service and some applications, but we'll save that for the next installment.

Conclusion

Your Raspberry Pi now has an accelerometer connected to it that stores the last minute's worth of data! This concludes our third installment of *How to Hack your Home with a Raspberry Pi*.

In the next installment, we will:

- Build a web service that reports the accelerometer data
- and build a PHP/JavaScript plotting application that:
  - uses the web service
  - and plots all three axes of the accelerometer data in realtime

Until next time, I hope you have fun playing around with your Pi!

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

9 *files:* https://phpa.me/digitalocean-systemd

Software History is Licensing

Chris Tankersley

The world of computers is an odd place. When I was younger, all we had was a Tandy Color Computer 2 because it was cheap. When we upgraded to an IBM running DOS, it was a major upgrade. I grew up during a time when the "family computer" was a common idea just because of cost. Now I walk around with a watch that is more powerful than most of those family computers, and it talks to my phone. And both of those integrate seamlessly with my tablet. The amount of computing power just around me as I type these words would have astounded seven-year-old me with that CoCo 2.

Software, too, has evolved. I started off with Microsoft Color BASIC on that old Tandy. Then the IBM let me experience DOS, and an upgrade opened up Windows 3.1. I never personally owned a modern Mac until I had a job in IT years later, but I used them throughout my time in school (and still want a gray iBook). I used a PC and Windows because that is what my dad bought.

Somewhere around 2000, I was at a book store and came across a boxed set for Linux Mandrake. I think it was something like fifty dollars, and I had disposable income. The box came with a bunch of CDs and some very hefty manuals. The manuals reminded me of the CoCo 2's manual, which went over all the hardware in the machine and even how to program in BASIC. I installed it on a second machine I had and was amazed.

I quickly ran into problems running it and had to search for help online. I started to learn about sharing source code, how to patch and recompile programs, and this whole world of sharing code. People gave out the source code to their software. For free. Legally. I did not have to pirate software. Which I never did. Piracy is bad.

This ecosystem was possible through what we now call Open Source Licensing. Software of all types came with a document that told me what I was allowed to do with the software, and normally that document said you cannot modify the software or share it, and technically you do not own the software.

The documents with Linux and the software I was exploring were different. I was allowed to modify and share the source code to software as long as I made my changes public. It all made sense. If something did not work correctly, I should be able to fix it and let other people know of the fix. I could not do that with Windows, Microsoft Office, or Photoshop on the Macs at school.

Why did we need those documents? Why was I allowed to muck around with these programs in Linux, but I was not allowed to do the same with most of the Windows and DOS software I had?

All Information Should Be Free

> *"All information should be free" — Steven Levy, "Hackers: Heros of the Computer Revolution," on the Hacker Ethics.*

In 1956, the Lincoln Laboratory designed the TX-0, one of the earliest transistorized computers. In 1958 it was loaned to MIT while Lincoln worked on the TX-2. The TX-0 amazed the early computer hackers at MIT. It did not use cards, and it was not cloistered away like the hulking behemoth of a machine from IBM that most people at MIT used for programming. You typed your program onto a ribbon of thin paper, fed it into the console, and your program ran.

Most importantly, the TX-0 was not nearly as guarded as the holy IBM 704 that the students and faculty normally used. That machine was governed by contracts and highly skilled system operators. You did not get to play with the IBM 704.

The TX-0 was different. Most of the hackers were free to do what they wanted with the machine. There was one problem, and it was somewhat of a large one — the TX-0 had no software. Like almost every piece of hardware that had come before it, software was something special to each device. The IBM 704 software would not run on the TX-0.

So the hackers at MIT created what they needed, and they shared that software freely and willingly. Computer software started as open.

Most of the software was kept in drawers, and when you needed something, you reached in and grabbed it. The best version of a tool would always be available, and anyone could improve it at any time. Everyone was working to make the computer and the software better for everyone else.

"All information should be free" was a core tenant of the hacker culture at MIT. No one needed permission to modify the software as everyone was interested in making the software, and thereby the TX-0, better.

As the machines changed and the software changed, this ethos did not. The software would be shared and changed to work on many different types of hardware, and improvements

were added over time. Needed the latest copy? Just ask for it. Need to fix it? Just fix it.

The ideals of the hacker culture at MIT did not change as it spread westward and as these computers invaded the lives of hobbyists. What changed was the business around computers, and like anything when it comes to humans, there was money to be made. These early computers still needed software written specifically for them, so users shared their code and ideas at Computer Clubs.

1975 and 1976 saw two important changes in the software landscape. The first was the commercialization of software. Ed Roberts, the "father of the personal computer" and the founder of MITS (Micro Instrumentation and Telemetry Systems), had decided not to give the Altair BASIC software to customers for free and instead charged $200 for the ability to write software.

"All information should be free" reared its head when the tape containing Altair BASIC disappeared from a seminar put on by MITS at Rickey's Hyatt House in Palo Alto, California. The birth of commercial, proprietary software almost immediately led to the second important change—the idea of pirating software.

For better or worse, copies of Altair BASIC started appearing and being shared. This did not sit well with Microsoft, which had licensed the software to Altair, and prompted a young Bill Gates to write "An Open Letter to Hobbyists"[1] about how not paying for Altair Basic was hurting his ability to pay software developers.

> *"Those who do not understand UNIX are condemned to reinvent it, poorly" — Henry Spencer.*

The 1970s also saw the development of the Unix operating system developed at AT&T by Ken Thompson, Dennis Ritchie, and others. AT&T, however, was not allowed to get into the computer business due to an antitrust case that was settled in 1958. Unix was not able to be sold as a product, and Bell Labs (owned by AT&T) was required to license its non-telephone technology to anyone who asked.

Unix was handed out with a license that dictated the terms of usage, as the software was distributed in source form. The only people who had requested Unix were those who could afford the servers, namely universities and corporations. The same entities that were used to just sharing software.

The open nature of Unix allowed researchers to extend Unix as they saw fit, much as they were used to doing with most software. They developed fixes and made improvements, folding them into mainstream Unix. While this eventually caused a nightmare when AT&T attempted to stop the University of California in Berkeley from distributing their own variant of Unix, we start to see the seeds of permissive licensing as the Berkeley Software Distribution releases with a very permissive license compared to the eventually heavily-commercial AT&T Unix license.

In 1980, copyright law was extended to include computer programs. Before then most software had freely been shared or sold on a good faith basis. You either released your software for everyone to use as public domain or sold it with the expectation that someone wouldn't turn around and give it away for free.

For all of his many faults, Richard Stallman was, and probably is, one of the last true Hackers from the MIT era. In a sort of hipster kind of way, he yearned for the time when software could be free, not shackled by laws or corporations. In a sense, software was meant to be shared and wanted to be shared. **"All information should be free."**

Stallman announced the GNU project in 1983, which was an attempt to create a Unix-compatible operating system that was not proprietary. NDAs and restricted licenses were antithetical to the ideals of free software that he loved.

The Free Software Foundation was founded in 1985, and along with it came the idea of "copyleft." In 1985 Stallman wrote the "GNU Manifesto," which detailed his thoughts on open source software and that software itself was meant to be free. Whether you agreed with it or not, the GNU Manifesto was a fundamental part of what we now consider Open Source.

Stallman then conglomerated his three licenses, the GNU Emacs, the GNU Debugger, and the GNU C Compiler, into a single license to better serve software distribution — the GPL v1, in 1989.

The release of the GPL, the release of a non-AT&T BSD Unix, and the flood of commercial software of the '80s and 90's lead us to where we are today and are the three major ideals that exist:

- Software should always be free — Copyleft
- Software should be easy to use and make the developers lives easier — Permissive
- Software should be handled as the creator sees fit — Commercial

The Open Source Initiative[2] helped codify what exactly it meant to be Open Source and which licenses truly embodied that idea. As long as you used one of their many suggested licenses, you could call your software legally Open Source.

Post Open Source

> *"younger devs today are about POSS — Post open source software. fuck the license and governance, just commit to github." — James Governor, @monkchips*

Up through the 2000s, when it came to software licensing, these three ideas covered almost all software. Due to companies and copyright, the legal verbiage provided by licenses like the MIT, BSD, GPL, and commercial licenses were needed. If it ever came to court, you needed to make sure you either explicitly approved of how software was shared or how you denied it.

1 *Hobbyists: https://en.wikipedia.*

2 *Initiative: https://opensource.org*

As time went on, the trend did seem to indicate that the development environment favored developer ease-of-use for code (permissive licensing) over a requirement of code sharing (copyleft). The general idea of permissive licensing was to make the Developer's life easier, but what if there was an even more permissive license than permissive licenses?

> *"Empowerment of individuals is a key part of what makes open source work, since in the end, innovations tend to come from small groups, not from large, structured efforts." — Tim O'Reilly*

As with everything, the internet changed how we shared code.

GitHub made it easy for anyone to throw code up on a website and made it easy to get that code down to your machine. For many years GitHub actively decided not to enforce a license on code uploaded as open-source repositories. GitHub left it up to the maintainer to sort that out. By 2016, 80–90% of projects did not bother with a license[3] even when in 2013 GitHub decided to start asking about licensing when new projects were created.

Eric Tachibana CC Attribution License

"All information should be free" has been a tenant of hackers since the 1960s. Instead of restricting the usage of code, why not just make it free? Completely free?

By the late 2000s and early-to-mid 2010s, a new trend was forming around the idea of releasing software under much more lax licenses, or even under what the US calls the Public Domain. In some extreme cases, code is being released without any license as to how it can be used, under the assumption that no license is the same as Public Domain.

The driving force behind this idea is "I don't care what you do with my code." It's a noble idea that hearkens back to the 1960s. Code does not need all of these rules around sharing and usage, just take my code and do what you want. The code on GitHub is no different than those old paper tapes sitting in a drawer for anyone to use.

There are even licenses that support this due to the way that copyright works. Licenses such as WTFPL (Do What the Fuck You Want to Public License) and the DBAD (Don't be a Dick) Public License, designed to get out of the nitty-gritty thinking when it comes to sharing code — here is code, just use it.

Post Open Source was a rebellion against the licensing structure of the past without a worry about the legal side. Just release code.

And Now, Today

Licensing has always been a murky pool of water, and the idea of Post Open Source, I feel, muddied the waters even further. Where we used to think about how to license a particular piece of software, we now either leave the license off, ignorant of why it was there in the first place, or just use whatever the first option Github offers.

A recent thread on Twitter was kicked up about WordPress, the GPL, and the licensing around plugins. If WordPress is GPLv2 and a plugin's PHP code must also be GPLv2[4], can a plugin author enforce other restrictions like saying you cannot distribute the software?

The opinion of WordPress, at least in 2009, was that no, a plugin's PHP code must be licensed under the GPLv2, and the GPLv2 forbids additional restrictions on the freedom of the software. You cannot license software under the GPLv2 and then tell someone they cannot modify or distribute the code.

As developing and distributing software has become easier, we run into an amount of ignorance. Developers do not understand what licensing is, which license may be appropriate for their goals, and the very real legal issues that can arise when using software against the stated license. Nearly 70 years of computer and software history is unknown to a huge number of developers.

Now that we have a good idea of how licenses evolved and why they are needed, next month, we will dive deeper into the actual licenses, what their implications are, and what might be appropriate for your project.

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

3 *license:* https://250bpm.com/blog:82/

4 *a plugin's PHP code must also be GPLv2:* https://wordpress.org/news/2009/07/themes

Understanding Supply Chain Security

Eric Mann

In the physical world, it's relatively easy to understand what a supply chain looks like—the security of physical goods in transit is a straightforward concept. This kind of security in the digital world can be harder to recognize but is just as critical.

In late January, news came out that a popular WordPress plugin and theme distributor was hacked[1]. AccessPress[2] publishes both free[3] and premium themes and plugins that power *hundreds of thousands* of websites.

You need to understand that this attack was in no way a simple breach. It's not that AccessPress shipped vulnerable code. They didn't publish a package that inadvertently permitted privilege escalation or remote code execution.

It appears a malicious actor *hacked AccessPress' servers* and used those servers to inject intentionally malicious code into updates of 93 separate packages—40 WordPress themes and 53 separate plugins.

This breach tricked WordPress sites into pulling down and installing code that would embed a backdoor into the site. The backdoor itself wasn't very sophisticated—reports indicate it would redirect viewers to spammy websites. Frankly, there are worse things a hacker can do with a backdoored site.

The backdoor itself

Affected themes and plugins shipped an additional `init.php` file bundled with an update. This file would modify WordPress core's `vars.php` file to insert the backdoor, then automatically delete itself to make things harder to detect.

In fact, if your site *were* infected, merely removing the impacted theme/plugin wouldn't fix the problem. You would also need to find and fix the infected `vars.php` file to remove the backdoor!

Luckily, researchers from both Sucuri[4] and Jetpack[5] have documented detailed instructions on how to clean an infected WordPress installation. The Jetpack team has also published a ruleset for the open-source YARA[6] tool to scan for a potential infection.

> *High-profile attacks against WordPress do little to fight against the long-held belief that WordPress itself is insecure. The reality is that any open-source tool that relies on automated updates from upstream vendors is potentially vulnerable to the same kind of attack. WordPress is a wildly popular platform, which produces an incredibly large attack surface—you hear more about WordPress-related hacks than other platforms because of the sheer number of WordPress-powered sites in the wild!*

Supply chain attacks

Most of the vulnerabilities that we worry about in software development impact our own code or deployments. We might forget to properly sanitize user input. A library we depend on is neglected and left sorely out of date. Someone on our team installed a testing utility to debug a problem and forgot to remove it.

These user mistakes are easy to make but also easy to catch. Supply chain attacks, however, are more indirect. Our team hasn't made a mistake and no one has directly attacked our product or platform.

The SolarWinds breach of February last year[7] was an extraordinary example of how things can go wrong with vendor dependencies. Rather than breach multiple applications, hackers managed to corrupt the SolarWinds server itself. From there, unsuspecting clients pulled down malicious updates and effectively invited the attackers into their infrastructure.

Such is the terrifying nature of a supply chain attack.

Any platform that pulls executable code from a third-party server is potentially subject to the same kind of attack. An attacker doesn't need to target your system directly. Pulling that code—updates, extensions, and the like—increases your attack surface.

Preventing a compromise

If you're reading this magazine, you probably write PHP. If you write PHP, you probably leverage Composer to manage dependencies. If you use Composer, you *still* don't have the support of cryptographic signatures[8] on the integrity of vendor updates. These are the kinds of protections in place on operating system packages and core extensions to various user applications we leverage in production.

1 *was hacked:* https://phpa.me/bleepingcomputer-wp

2 *AccessPress:* https://accesspressthemes.com

3 *free:* https://wordpress.org/plugins/search/accesspress/

4 *Sucuri:* https://blog.sucuri.net/?p=28341

5 *Jetpack:* https:\/\/wp.me\/p1moTy-ACL

6 *YARA:* https://github.com/VirusTotal/yara/

7 *last year:* https://phpa.me/supply-chain-security

8 *signatures:* https://github.com/composer/composer/issues/4022

While there might not yet be cryptographic security around modules written in PHP, the core team leverages GPG signatures[9] to verify the integrity of updates shipped for the language itself.

Until we have strong, cryptographically verified protection on vendor updates, it's critical that your dev team manually inspect and approve update packages. I'm not saying we should never install updates—most updates are going to be fine and actually improve the security and stability of your system.

Do not view issues like SolarWinds or the recent Access-Press breach as justification for skipping or blocking automatic updates. Please keep installing these updates immediately (or automatically) when they're released. However, you *must* take time to review the content of these updates to ensure nothing snuck in.

Read the release notes. Check a code diff. Ensure your team is verifying both the trustworthiness of its own code and that of any code shipped by a third party.

9 *leverages GPG signatures:* https://www.php.net/gpg-keys.php

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

Related Reading

- *Security Corner: Supply Chain Security* by Eric Mann, February 2021. https://phpa.me/supply-chain-security
- *Community Corner: My Picks From Packagist* by James Titcumb, May 2018. https://phpa.me/community-corner-may-2018

Queues with Horizon

Joe Ferguson

In the January 2022 edition, my friend Chris Tankersley wrote Education Station: Background Queues[1], a fantastic primer for using background queues, or workers with your PHP application. This month we will implement Laravel Horizon[2], a dashboard monitor for your Redis queues.

The very basics of using queues in modern web applications are to offload long-running jobs or processes which may take longer than an HTTP request may allow. The primary benefit of queueing this work rather than your application making the user wait is to add perceived speed.

Traditionally, the task of sending an email requires PHP to connect to a remote mail system to relay the message. While this can be a quick transaction, it can fail if the remote system doesn't respond, but if we queued the mail sending portion of this work, we could send the user back to the application to continue what they were doing. The perceived speed improvement is the user is no longer waiting on the mail to be processed as they're immediately redirected and continue their workflow. This use case is well documented in modern frameworks such as Laravel[3], Symfony[4], and CakePHP[5].

Another use case for queueing would be for processing users' file uploads—parsing and saving CSV data or pushing file uploads to cloud storage. A third example of when to queue work would be compiling a complex report where the query time may be longer than a few seconds. If we're sending mail, parsing or uploading files, or running long reporting jobs, queues allow us to easily organize and process this work without slowing down our users.

Using Laravel 9, the newest release, we will create a new Laravel application, add the Predis[6] library for Redis, and install Laravel Horizon. The following commands will replicate the application in your local development environment using PHP 8.0.

```
composer create-project laravel/laravel horizon
composer require predis/predis ~1.0
composer require laravel/horizon
php artisan migrate
php artisan horizon:install
php artisan make:job ParseAndSaveCsv
php artisan make:job SaveUploadToS3
php artisan make:job GenerateReport
```

1 *Background Queues:* https://phpa.me/background-queues
2 *Laravel Horizon:* https://laravel.com/docs/8.x/horizon
3 *Laravel:* https://laravel.com/docs/9.x/mail#queueing-mail
4 *Symfony:* https://symfony.com/doc/current/messenger.html
5 *CakePHP:* https://phpa.me/cake-queue-mailer
6 *Predis:* https://github.com/predis/predis

Laravel Configuration

We need to add a database connection for our Redis service in `config/database.php`:

Listing 1.

```
...
'connections' => [
    'redis' => [
        'driver' => 'redis',
        'connection' => 'default',
        'queue' => 'default',
        'retry_after' => 90,
        'block_for' => 5,
    ],
...
```

We'll also want to inspect `config/horizon.php` to note we have `environments` configured as the defaults for `local`; this will give us 2 processes at most, but at least one. We're setting these values intentionally low to simulate real-world delays you might see in production. You will want to use larger values here in production.

In our `.env` configuration, we need to ensure we've configured `QUEUE_CONNECTION=redis`; otherwise, our application will use the default `sync` driver instead of Redis.

Creating and Dispatching Jobs

Earlier we created three jobs: `ParseAndSaveCsv`, `SaveUploadToS3`, `GenerateReport`. These will be the jobs we queue within our application. Each of our job classes consists of the same boilerplate, manually tagging our jobs for Horizon and the `handle()` method, which for our examples, will simply

Listing 2.

```
...
'environments' => [
    'local' => [
        'supervisor-1' => [
            'minProcesses' => 1,
            'maxProcesses' => 2,
        ],
    ],
...
```

Listing 3.

```
1. class ParseAndSaveCsv implements ShouldQueue
2. {
3.     use Dispatchable,
4.         InteractsWithQueue,
5.         Queueable,
6.         SerializesModels;
7.
8.     public function tags()
9.     {
10.        return ['name', 'ParseAndSaveCsv'];
11.    }
12.
13.    public function handle()
14.    {
15.        Log::info('Running Job ParseAndSaveCsv');
16.        sleep(15);
17.    }
18. }
```

log and sleep for a while. Sleeping for a while will give us realistic numbers in our dashboard instead of all of the jobs instantly completing because everything is running in our local environment. An example job is shown in Listing 3.

To generate a number of our jobs to simulate workloads, we can use Listing 4 to add several jobs to the queue with different delay settings. Adding a delay to the `dispatch()` call in this use case instructs the workers to not process the job until after the delay time has passed.

Now we're ready to run Horizon. If you are already running `queue:work` commands, you no longer need to run them as the `php artisan horizon` command will run our workers as we've configured.

Horizon's value is the dashboard it provides out of the box and the ability to see pending, completed, and failed jobs. From the dashboard, we can see we've had 92 jobs per minute, 7,500 jobs in the past hour with a max wait time of 8 minutes. We get the max wait time because our jobs will `sleep` for a

Listing 4.

```
1. for ($i = 1; $i <= 100; $i++) {
2.     ParseAndSaveCsv::dispatch()->delay(now()->addMinutes(1));
3.     SaveUploadToS3::dispatch()->delay(now()->addMinutes(1));
4.     GenerateReport::dispatch()->delay(now()->addMinutes(1));
5.
6.     ParseAndSaveCsv::dispatch()->delay(now()->addMinutes(2));
7.     SaveUploadToS3::dispatch()->delay(now()->addMinutes(2));
8.     GenerateReport::dispatch()->delay(now()->addMinutes(2));
9.
10.    ParseAndSaveCsv::dispatch()->delay(now()->addMinutes(3));
11.    SaveUploadToS3::dispatch()->delay(now()->addMinutes(3));
12.    GenerateReport::dispatch()->delay(now()->addMinutes(3));
13. }
```

short period and because we are only processing 2 jobs at a time from our `maxProcesses` configuration, which is also shown as Total Processes.

After more time passes and scheduling more jobs, we can see our Max Wait Time has increased to 38 minutes and jobs per minute decreased to 48.

Figure 2. Horizon dashboard over time.

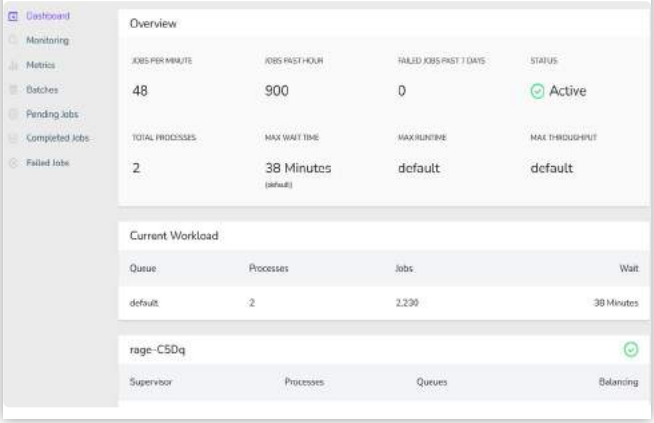
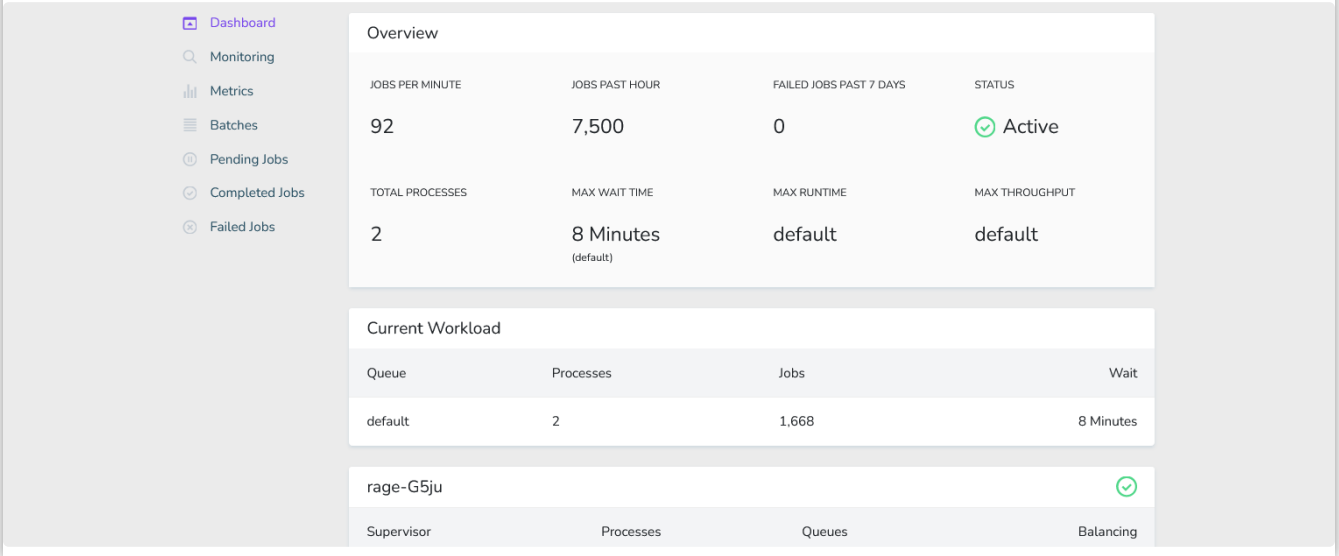


Figure 1. Horizon Dashboard overview



Viewing Pending Jobs in Horizon (Figure 3) allows us to see the queue, tags, and time the job was queued.

We can click into the Jobs and see a breakdown with the complete job information such as Pushed At, Delayed Until, and Completed At dates which give us insights into how long our job took to complete. See Figure 3. This information helps us debug how many workers or processes we should have so that our jobs are flowing through the system without any bottlenecks. As we scale our application, we scale our queue workers as well to handle the larger volume of jobs to ensure jobs happen efficiently and without needlessly waiting.

Horizon allows us to pause at any time by running `php artisan horizon:pause`, which will return "Sending USR2 Signal To Process: 10610," and the Horizon Status in the top right of the dashboard will change to "Status Paused." Once we're ready to resume processing, we run `php artisan horizon:continue`, which informs us "Sending CONT Signal To Process: 10610," and the logs from our `php artisan horizon` will continue:

```
[0ffe17] Processed:  App\Jobs\ParseAndSaveCsv
[f23671] Processing: App\Jobs\GenerateReport
[5dd0ed] Processed:  App\Jobs\GenerateReport
[be7fe7] Processing: App\Jobs\SaveUploadToS3
[f23671] Processed:  App\Jobs\GenerateReport
[b937eb] Processing: App\Jobs\GenerateReport
[be7fe7] Processed:  App\Jobs\SaveUploadToS3
[22f9de] Processing: App\Jobs\SaveUploadToS3
```

Figure 3. Pending jobs in Laravel Horizon

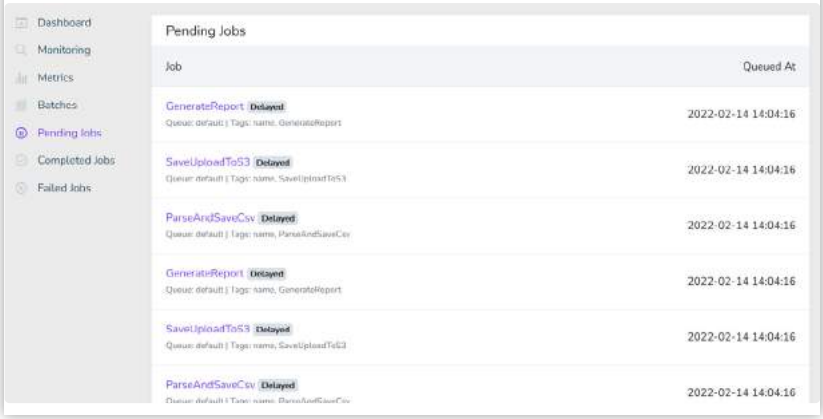
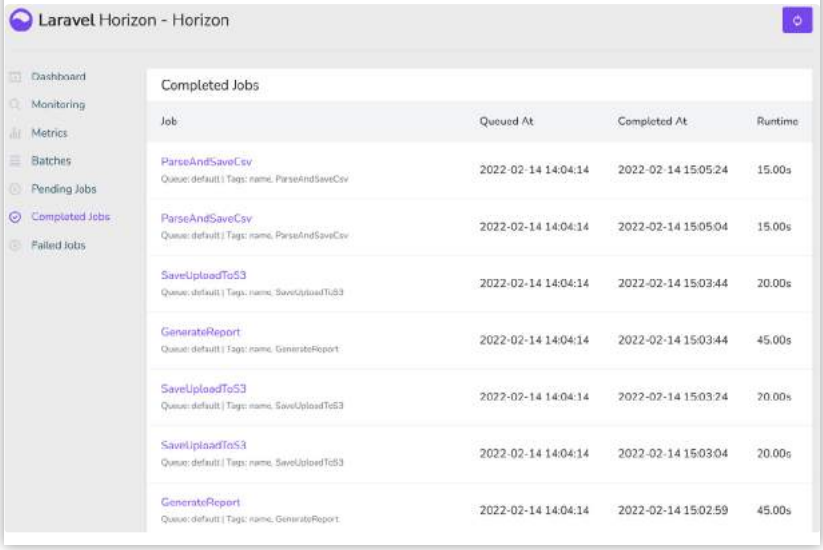


Figure 4. Completed Jobs in Laravel Horizon



Setting Your Horizon

If you aren't already using Queues in your application, we've covered enough to get you going and also have insights into your queue via Laravel Horizon. Before you take off into production, make sure you review Deploying Horizon[7]. You must restrict the Horizon dashboard to at *least* logged-in users or an administrator user group as the dashboard will contain sensitive data about your jobs. You'll also want to use a process monitor such as Supervisor[8] to keep the worker processes running. A basic Supervisor configuration for our application would be found at `/etc/supervisor/conf.d/horizon.conf` and might look like Listing 5.

An important note about `stopwaitsecs` is it should be greater than the number of seconds that your longest job requires to run; otherwise, Supervisor may kill your job before it's had a chance to finish processing.

Listing 5.

```
1. [program:horizon]
2. process_name=%(program_name)s
3. command=php /path/to/application artisan horizon
4. autostart=true
5. autorestart=true
6. user=applicationuser
7. redirect_stderr=true
8. stdout_logfile=/path/to/logs/horizon.log
9. stopwaitsecs=3600
```

Before we start supervisor, we should reread and update the configuration:

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start horizon
```

Here is another tip about Queues and Laravel Horizon, which might save you some time, grief, or money one day. Your jobs are going to fail; something will eventually go wrong, especially if you're making outbound API calls in Jobs

7 *Deploying Horizon:* https://laravel.com/docs/9.x/horizon#deploying-horizon

8 *Supervisor:* http://supervisord.org/configuration.html

to populate data for your application from a 3rd party. Maybe this is a highly specialized API where you incur a cost per API call, so you mustn't fail the job and retry it a bunch of times, racking up your API bill. To prevent your jobs from being attempted too many times, you can set the `$tries` property to a low value to ensure you only retry the job a minimal amount of times before it is marked as failed. You can also specify `$maxExceptions` which will allow up to this many unhandled exceptions.

```
public $tries = 5;
public $maxExceptions = 3;
```

This setting could be useful for integrating with unreliable data sources that may not always be stable. If your requests are time-based[9], you can also utilize the `retryUntil()` method. Furthermore, preventing job overlaps[10] using `WithoutOverlapping()` helps if a job is modifying a model or changing state when only one Job should be performing this update at a time.

If you are dealing with a very unreliable 3rd party, your first stop in your job configurations should be Throttling Exceptions[11]. To throttle exceptions coming from a job, we can apply the following configuration:

Listing 6.

```
1. use Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis;
2.
3. public function middleware()
4. {
5.     return [new ThrottlesExceptionsWithRedis(5, 30)];
6. }
7.
8. public function retryUntil()
9. {
10.    return now()->addMinutes(30);
11. }
```

The `ThrottlesExceptions(5, 30)` instructs the framework to allow 5 exceptions thrown by the job before being throttled for 30 minutes. If there haven't been 5 exceptions, the job will be returned to the queue to be retried nearly immediately. To prevent an immediate retry, we can add `->backoff(5)]` to delay the retry attempt by 5 minutes. If you end up using these methods, ensure you use the `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis` middleware instead of the default `ThrottlesExceptions`.

Now you're ready to start implementing Horizon in your application; happy queueing!

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

Related Reading

- *Boosting User Perceived Performance with Laravel Horizon* by Scott Keck-Warren, August 2021. http://phpa.me/boost-performance
- *Education Station: Background Queues* by Chris Tankersley, January 2022. https://phpa.me/background-queues
- *Background Processing & Concurrency With PHP* by Matthew Schwartz, January 2018. https://phpa.me/concurrency-with-php
- *Education Station: Producer-Consumer Programming* by Edward Barnard, September 2018. http://phpa.me/education-sept-18

9 time-based: https://laravel.com/docs/9.x/queues#time-based-attempts

10 preventing job overlaps: https://laravel.com/docs/9.x/queues#preventing-job-overlaps

11 Throttling Exceptions: https://laravel.com/docs/9.x/queues#throttling-exceptions

Better Late Than Never

Edward Barnard

Last month, we created "role-based lookups" with "When You Know the Pattern." Database-related code can be difficult to unit test, and once unit tests are written, they can be difficult and time-consuming to maintain. This month we'll carefully look at this problem and possible solutions. We'll walk through the test suite after extensive refactoring to keep it clear and expressive.

Are Tests Needed?

"When You Know the Pattern[1]" in February 2022 php[architect] walked through my "role-based lookup" class. I did not show you any automated tests supporting or verifying that class. That's because there were no tests at all.

I had a reason for writing no tests, and I later realized this was a mistake on my part. This month we will look at my reasons for not writing automated tests. Then we'll walk through what I did write.

Database Layer

In general, I have found that small database-specific methods don't break. For example, I might write a finder method that determines the type of safety certification required for members of a given team. Once the method is written, and in production, things like that generally don't break. It's extracting a specific column from a specific table based on specific query criteria.

What if we add a new feature to our codebase? That feature is unlikely to break that database query. Nor is the table going to behave differently during, for example, a peak-usage period.

Of course, if we change the table or a new business requirement changes that use case, that's different. At that point, I would expect to change the query. But an automated test isn't going to affect that situation.

Generally speaking, then, I do not write tests for "finder" methods or other similarly trivial queries. The likelihood of a regression suddenly appearing is low. Use that potential time investment for testing higher-level features.

The other difficulty with testing database-specific methods is the ongoing time to keep test data up to date. Furthermore, tests that use an actual database run far more slowly.

All in all, experience has shown me that unit tests for small low-level "finder" and "updater" methods are not worth the ongoing time investment. To be sure, if regressions do start appearing, that's a different situation. Investigate and consider whether regression tests are the correct way to help prevent future problems.

Role-based Lookup

What's different about our role-based lookup introduced in "When You Know the Pattern"? The difference is that the class is 539 lines long. That means it's far more than a trivial "finder" method!

The class has multiple responsibilities. I don't mean in the sense of violating the Single Responsibility Principle[2]. I mean in the sense of having several things to do in accomplishing its primary reason for existence.

- We make the lookup easily accessed anywhere in our codebase (Registry pattern).
- We cache a copy of the lookup results (Memoization pattern).
- We provide results from the cache when available.
- We report missing records through an exception report.
- We consider the possibility of a missing record and return the appropriate result.
- We support multiple "hops" from one memoized record to another.

The class has 28 public methods. That in itself is a warning that this class might be taking on too much for one class! Martin Fowler's *Refactoring: Improving the Design of Existing Code*[3], second edition, includes a catalog of "code smells."

He describes "Large Class" as a "code smell" on page 82. After describing the tell-tale "code smells" of too many instance variables or too much code, he prescribes:

> *The clients of such a class are often the best clue for splitting up the class. Look at whether clients use a subset of the features of the class. Each subset is a possible separate class. Once you've identified a useful subset, use "Extract Class", "Extract Superclass", or "Replace Type Code with Subclasses" to break it out.*

1 *When You Know the Pattern:* https://phpa.me/2022-02-ddd

2 *Single Responsibility Principle:* https://w.wiki/yHV

3 *Existing Code:* https://martinfowler.com/books/refactoring.html

At this moment, Fowler's advice does not yet apply. The role-based lookup supports the season-registration workflow. That workflow, as a whole, is its client. It's actually well-focused on that one specific purpose. Thus Fowler's description assures us that we are on the right track–for the moment.

However, I will likely need to do role-based lookups of different information during future development. At that point, the two different workflows represent Fowler's "clients of such a class," and there may be a good reason for refactoring.

The above list of bullet points (Registration pattern, Memoization pattern, etc.) tells me that unit tests might be a good idea. The possibility of later refactoring also tells me unit tests will be a good idea. If you've ever tried to refactor several hundred lines of code without tests, you'll know what I mean! Refactoring with tests in place is a far more pleasant experience.

However, there's one more reason for adding unit tests. That single class for role-based lookups quickly became central to the entire season-registration workflow. I would not want it to accidentally break in the middle of season registration!

Let's recap to this point. I was happily implementing that season-registration workflow. I was confident in my decision not to add automated tests to the low-level database-access class. The role-based lookups are most definitely trivial "finder" methods and thus meet my criteria for not investing the time to write a test suite.

Then came the problem.

I happened to notice a coding error in that "role-based lookups" class as I was implementing a step in the season-registration workflow. I fixed the error and added a task to our ticket system to do a full desk check[4] of the class looking for any similar errors.

When I got to that "desk check" task, I realized everything we just discussed above. I decided this class is exactly the sort of place that needs an automated unit-test suite. I discovered a second coding error as I was developing the test suite. Eventually, I would have found the error. But it's definitely more pleasant to find and remove the error before the code reaches production.

A perfect example of "better late than never!"

Other People's Experience

My first goal was to figure out a way to test the role-based lookups without actually using a database. We need to make a careful distinction, that is, a careful separation of what we want to test and what we do not need to test.

Whoops! Wait! Let's back up. How do I know this "first goal" is the right approach? I could simply claim that I know this from experience. But what I do know from experience is that it's far less painful to learn from other people's experiences! There's a famous saying, "experience is what you get right after you needed it."

This is why I quote from books such as Fowler's *Refactoring* or the Gang of Four's *Design Patterns*[5]. These are among my sources of experience. We saw last month that if you know the pattern, Domain-Driven Design enables you to easily implement the pattern within our modern PHP ecosystem.

The key to making that work is to carefully separate the parts of code that are closely coupled to your PHP framework of choice from the parts of code that need not be coupled to the framework. That's why I expressed "my first goal" in terms of making exactly that distinction or separation.

Sure enough, Gerard Meszaros' *xUnit Test Patterns: Refactoring Test Code*[6] has a pattern for us.

> *Here is something odd. The list price on my copy (next to the bar code on the back cover) is $64.99 (fourth printing of first edition, September 2010). Amazon's "look inside" feature (accessed 12 February 2022) shows the price as $59.99 (third printing of first edition, February 2009). What's odd is that Amazon shows the selling price of a new copy as $110 and that it ships from the UK. Used prices begin at $135. I'm not sure if this is a reflection of the book's actual value or Amazon's price gouging in support of the $47.99 Kindle price.*
>
> *Generally speaking, when I notice something odd, I flag it–because it may lead to deeper understanding later. Thus this is an observation without a conclusion!*

xUnit Test Patterns pages 263-264 describes the "project smell" of "Developers Not Writing Tests." Meszaros presents three causes:

- Not enough time
- Hard-to-test code
- Wrong test automation strategy

The first cause, "not enough time," is self-explanatory. We've all been there. For "hard-to-test code," Meszaros explains:

> *A common cause of developers not writing tests, especially with legacy software (i.e., any software that doesn't have a complete suite of automated tests), is that the design of the software is not conducive to automated testing.*

At the top of this discussion, I basically depended on a reason not to write automated tests. Meszaros reminds us to look for any excuse at all to write automated tests. Over time, his outlook will cause your project to become more supple, more stable, more open to automated tests–and these test suites, in turn, support adding new features without fear of breaking something already running in production.

What can we learn about the third cause, "wrong test automation strategy?"

4 *desk check:* https://phpa.me/techopedia-desk-check

5 *Design Patterns:* https://phpa.me/amazon-design-patterns

6 *Test Code:* https://phpa.me/amazon-test-patterns-refactoring

> *Another cause of developers not writing tests may be a test environment or test automation strategy that leads to "Fragile Tests" or "Obscure Tests" that take too long to write. We need to ask the five why's to find the root causes. Then we can address those causes and get the ship back on course.*

Our long-winded discussion has done just that. We have dug into the root causes and concerns to get "the ship back on course."

Obscure Test

I know from my own experience that it's difficult to step away from a database dependency. That's the challenge we're dealing with here as we consider a test suite for our "role-based lookups."

Have you ever been in a design discussion or problem-solving discussion where two separate issues get conflated into the same discussion? It can border on hilarious to observe an argument where two people use the same word to mean two vastly different things. Neither realizes the other is talking about something else entirely.

Domain-Driven Design (DDD) tackles this problem head-on with its foundational concepts of "ubiquitous language" and "bounded context." Rather than conflating the concepts and causing confusion or misunderstanding, DDD aims for separation and clarity.

We're looking at the same principle here but on a smaller scale. Within our "role-based lookups" class, we need to carefully separate the "role-based lookup" features (the bullet list above) from the framework-specific mechanics of database access (the literal "finder" methods).

Why? This separation allows us to clearly delineate what we are testing and what we are not testing. Meszaros explains with "Obscure Test," page 186:

> *Automated tests should serve at least two purposes. First, they should act as documentation of how the system under test should behave; we call this "tests as documentation." Second, they should be a self-verifying executable specification. These two goals are often contradictory because the level of detail needed for tests to be executable may make the test so verbose as to be difficult to understand.*
>
> *The first issue with an Obscure Test is that it makes the test harder to understand and therefore maintain. It will almost certainly preclude achieving "tests as documentation," which in turn can lead to "high test maintenance cost."*

> *The second issue with an Obscure Test is that it may allow bugs to slip through because of test coding errors hidden in the Obscure Test. This can result in "buggy tests." Furthermore, a failure of one assertion in an "eager test" may hide many more errors that simply aren't run, leading to a loss of test debugging data.*

Experience tells me the Obscure Test discussion is exactly the sort of problem that crops up when writing tests for database-related code. When using test fixtures[7] (pre-defined values to load in a test database for the specific test), the test fixture manipulation tends to obscure what we're actually trying to test.

On the other hand, when avoiding test fixtures by resorting to test doubles[8] or other forms of mock objects, we have the same problem. Test setup and teardown are so complicated that it's hard to understand the actual intent of the unit test.

Meszaros comes through with our necessary guideline:

> *The root cause of an Obscure Test is typically a lack of attention to keeping the test code clean and simple. Test code is just as important as the production code, and needs to be refactored just as often.*

We now have a sort of catch-22[9] situation. We're suffering from analysis paralysis[10]. We know it's difficult to deal with database dependencies. How will we do that and keep the test code clean and simple?

Kent Beck with *Test-Driven Development: By Example*[11] has the prescription on page 11:

> *Write a test. Make it run. Make it right.*
>
> *The goal is clean code that works. Clean code that works is out of the reach of even the best programmers… Divide and conquer. First, we'll solve the "that works" part of the problem. Then we'll solve the "clean code" part. This is the opposite of architecture-driven development, where you solve "clean code" first, then scramble around trying to integrate into the design the things you learn as you solve the "that works" problem.*

Test First and Last Name

The system under test (Meszaros, p. 810) includes `userFirstName()` and `userLastName()` in Listing 1:

7 *test fixtures:* https://phpa.me/phpunit-docs-sharing-fixture

8 *doubles:* https://phpunit.readthedocs.io/en/9.5/test-doubles.html

9 *catch-22:* https://w.wiki/MxK

10 *analysis paralysis:* https://en.wikipedia.org/wiki/Analysis_paralysis

11 *By Example:* https://phpa.me/amazon-TDD-Kent-Beck

Listing 1 includes two `public` methods and one `private` method. Both public methods include the annotation `/** covered */`. I used this annotation as a checklist while writing the test suite to ensure I had full coverage of all public methods.

I intended to delete those comments once the test suite was written. However, they don't hurt anything, and we already know that we'll likely be enhancing and/or refactoring this class in the future. I decided to leave the comments in place as a reminder to "future me" to ensure full test coverage at that time.

Listing 1.

```
1. /** covered */
2. public function userFirstName(): string
3. {
4.     $this->loadUser();
5.     if (null == $this->user) {
6.         return '';
7.     }
8.     return $this->user->first_name;
9. }
10.
11. /**
12.  * User row cannot be missing because we begin with a valid UserRole
13.  * which has a foreign key constraint on users
14.  *
15.  * @return void
16.  */
17. private function loadUser(): void
18. {
19.     if (null !== $this->user) {
20.         return;
21.     }
22.     $this->user = $this->usersTable->get($this->userRole->user_id);
23. }
24.
25. /** covered */
26. public function userLastName(): string
27. {
28.     $this->loadUser();
29.     if (null == $this->user) {
30.         return '';
31.     }
32.     return $this->user->last_name;
33. }
```

Listing 2.

```
1. class UserTest extends MockeryTestCase implements CFixtureRoleBasedLookup
2. {
3.     use ParticipantTrait;
4.
5.     private UserRole $userRole;
6.
7.     protected function setUp(): void
8.     {
9.         parent::setUp();
10.
11.         $this->loadUsersTable();
12.         $this->loadUserRolesTable();
13.
14.         $user = $this->usersTable->newEntity(self::USER_1);
15.         FixtureReporter::reset();
16.        /** @var UsersTable|\Mockery\LegacyMockInterface|\Mockery\MockInterface $usersTable */
17.         $usersTable = Mockery::mock(UsersTable::class)->makePartial();
18.         $usersTable->shouldReceive('get')->once()->andReturn($user);
19.
20.         $this->userRole = $this->userRolesTable->newEntity(self::USER_ROLE_1);
21.         $instance = FixtureReporter::getInstance($this->userRole);
22.         $instance->setUsersTableMock($usersTable);
23.     }
24. }
```

Listing 2 shows the first few lines of the "first and last name" test suite.

Our class extends `MockeryTestCase`. The Mockery[12] library includes PHPUnit[13] integration through `MockeryTestCase`. The key point to remember when using Mockery's "expectations" feature is to check those expectations during the test "tear down." `MockeryTestCase` handles that detail for you.

Note the constant `self::USER_1` around line 14 of Listing 2. That's a hard-coded test fixture. In Listing 3, my whimsical values represent a single row of the `users` database table. Our testing strategy will be to pre-load that database row and verify that "first and last name" results match our test fixture.

Note that `setUp()` in Listing 2 makes no reference to `RoleBasedLookup` which is our system under test (SUT). Rather than `RoleBasedLookup::getInstance()` the test calls `FixtureReporter::getInstance()`. We're using a "test double" (Meszaros, page 522):

> *How can we verify logic independently when the code it depends on is unusable? How can we avoid "slow tests"? We replace a component on which the SUT depends with a "test-specific equivalent."*

Our test double is Listing 4. Take a look then I'll explain the odd class name.

The odd class name `FixtureReporter` comes as a result of refactoring. I had a whole collection of test doubles and

12 *Mockery:* https://phpa.me/mockery-doc-phpunit

13 *PHPUnit:* https://phpa.me/phpunit-doc-index

Listing 3.

```
1. public const USER_FIRST_NAME = 'Neithhotep';
2. public const USER_LAST_NAME = 'Narmer';
3. public const USER_EMAIL = 'nn@example.com';
4. public const USER_PASSWORD = 'Menes1stDynasty?';
5. public const USER_PIN = '3100';
6. public const USER_USERNAME = 'Menes3100BC';
7. public const USER_1 = [
8.     User::FIELD_EMAIL_ADDRESS => self::USER_EMAIL,
9.     User::FIELD_USER_NAME => self::USER_USERNAME,
10.    User::FIELD_FIRST_NAME => self::USER_FIRST_NAME,
11.    User::FIELD_LAST_NAME => self::USER_LAST_NAME,
12.    User::FIELD_PASSWORD => self::USER_PASSWORD,
13.    User::FIELD_PIN => self::USER_PIN,
14.    User::FIELD_CREATED_BY => self::CREATED_BY,
15.    User::FIELD_MODIFIED_BY => self::MODIFIED_BY,
16. ];
```

realized I could make the intent far more clear by folding the "test double" support into a single class. I haven't come up with a name as yet that applies to all of the tests it supports, so I'm leaving the odd name in place until I do.

We already saw ("When You Know the Pattern" section "Model Access As Trait" and its Listing 2) that the framework-specific Model classes get loaded as protected properties of `RoleBasedAccess`. They load during `getInstance()` execution.

Listing 4's `getInstance()` invokes the parent `getInstance()`, that is, the SUT (system under test). Then, with `setUsersTableMock()`, our test double allows the test program to replace the framework's Model class with our carefully-crafted "mock object" (Meszaros, p. 544):

> *How do we implement Behavior Verification for indirect outputs of the SUT? How can we verify logic independently when it depends on indirect inputs from other software components? We replace an object on which the SUT depends with a test-specific object that verifies it is being used correctly by the SUT.*

Listing 2 accomplishes this task by setting the `users` and `user_roles` table models to return our pre-built "user" and "user role" entities. This process allows us to verify that our SUT does extract the correct fields from the correct entities.

Listing 5 shows the complete "first and last name" test suite—the remainder of the class setup with Listing 2.

Even with so much happening "behind the scenes," so to speak, our tests remain clear and expressive. We accomplished the twin goals (Meszaros, p. 186, "Obscure Test"):

Listing 5.

```
1. public function testFirstName(): void
2. {
3.     $lookup = FixtureReporter::getInstance($this->userRole);
4.     $expected = $lookup->userFirstName();
5.     $actual = self::USER_FIRST_NAME;
6.     self::assertSame($expected, $actual);
7. }
8.
9. public function testFirstNameCached(): void
10. {
11.     $lookup = FixtureReporter::getInstance($this->userRole);
12.     $expected1 = $lookup->userFirstName();
13.     $expected2 = $lookup->userFirstName();
14.     self::assertSame($expected1, $expected2);
15. }
16.
17. public function testLastName(): void
18. {
19.     $lookup = FixtureReporter::getInstance($this->userRole);
20.     $expected = $lookup->userLastName();
21.     $actual = self::USER_LAST_NAME;
22.     self::assertSame($expected, $actual);
23. }
```

Listing 4.

```
1. class FixtureReporter extends RoleBasedLookup
2. {
3.     public static function getInstance(UserRole $userRole): self
4.     {
5.         /** @noinspection PhpUnnecessaryLocalVariableInspection */
6.         /** @var \App\Test\Fixture\BoundedContexts\Infrastructure\GlobalLookup\FixtureReporter $instance */
7.         $instance = parent::getInstance($userRole);
8.         return $instance;
9.     }
10.
11.     public function setUsersTableMock(UsersTable $mock): void
12.     {
13.         $this->usersTable = $mock;
14.     }
15. }
```

- Demonstrate how the system under test should behave, that is, provide a clear example of intended usage.
- Provide a self-verifying executable specification.

We avoided the two concerns:

- Obscure tests are harder to understand and therefore maintain.
- Obscure tests may allow bugs to slip through because of test coding errors hidden in the obscure test.

Listing 6.

```
1. public function testSeasonParticipantConfirmation(): void
2. {
3.     $lookup = FixtureReporter::getInstance($this->userRole);
4.     $this->persistTeam();
5.     $this->persistLeague();
6.     $this->persistSeason();
7.     $this->persistParticipant();
8.     $this->persistSeasonParticipantConfirmation();
9.
10.    self::assertNotNull($lookup->seasonParticipantConfirmation());
11. }
```

Listing 7.

```
1. protected function persistTeam(): void
2. {
3.     $lookup = FixtureReporter::getInstance($this->userRole);
4.     $this->loadTeamsTable();
5.     $team = $this->teamsTable->newEntity(self::TEAM_1);
6.     $team->id = self::PERSIST_TEAM_ID;
7.     /** @var TeamsTable|\Mockery\LegacyMockInterface|\Mockery\MockInterface $teamsTable */
8.     $teamsTable = Mockery::mock(TeamsTable::class)->makePartial();
9.     $teamsTable->shouldReceive('get')->once()->with(103)->andReturn($team);
10.    $lookup->setTeamsTableMock($teamsTable);
11. }
```

Listing 8.

```
1. public function testMissingLeagueRecord(): void
2. {
3.     $this->persistTeam();
4.
5.     $lookup = FixtureReporter::getInstance($this->userRole);
6.     /** @var LeaguesTable|\Mockery\LegacyMockInterface|\Mockery\MockInterface $leaguesTable */
7.     $leaguesTable = Mockery::mock(LeaguesTable::class)->makePartial();
8.     $leaguesTable->shouldReceive('get')->once()->andThrow(RecordNotFoundException::class);
9.     $lookup->setLeaguesTableMock($leaguesTable);
10.
11.    $actual = $lookup->leagueId();
12.    $expected = 0;
13.    self::assertSame($expected, $actual);
14. }
```

Multi-hop Tests

Because of their greater number of dependencies, the multi-hop tests required some careful refactoring.

Listing 6's general flow is similar to the "first and last name" tests we saw in Listing 5. I refactored the setup into a parent class. Listing 7 shows `persistTeam()`.

The one key difference is that in "hopping" from one memoized record to another, we assume the record has actually been saved and that, therefore, we can look it up by its primary key. Therefore the mocking process needs to incorporate the primary-key lookup.

Take another look at Listing 6. The test's setup flow and the intent of the test should be clear. We know from the `RoleBasedLookup::seasonParticipantConfirmation()` method signature (not shown) that the method returns either a valid object of the correct class or `null`. Thus it should be clear that our test is verifying that, given the records exist for each hop, we do, in fact, produce the correct record.

Missing Records

Our "role-based lookups" also handle error conditions. Here's how we can verify an error situation. Here are the relevant business rules:

- Finding the "league" record requires first finding the "team" record and using the team record to find the league record.
- When requesting the league ID, if the league record is missing, return `0` as the league ID.
- When the league record is missing, report this fact in the exception report.

Listing 8 tests for the league ID being zero when the league record is missing.

With our test in Listing 8, we ensure the team record does exist, but the league record does not. This test is closely coupled to our PHP framework in use. With this particular framework (CakePHP 4.x), the Model class's `get()` method throws a `RecordNotFoundException` when the record is

missing. Therefore our test throws the necessary exception and then verifies that we got a result of zero.

We could compress the last three lines of the test (actual, expected, assert same) into a single line of code. However, I felt it was more clear to carefully lay out what is really expected, how we get it ("actual"), followed by the actual validation (assert same). That's because the setup is a bit convoluted and detracts from the clarity of the test.

Should I have extracted that convoluted setup into a separate class with an expressive name? Arguably, yes. But this particular 200-line class only tests the "missing record" edge cases. Extracting each "missing record" edge case into its own separate method made the whole situation far more difficult to follow.

Instead, I pulled all of the "missing record" tests into a single class to read from top to bottom (if you cared to). Each test follows a similar structure–and I took care that each case clearly shows that same structure or sequence.

There's one more piece to the puzzle. Our other requirement is that the role-based lookup creates an exception report when a table record is missing, and it definitely should not be missing. Since that's the case for all "missing record" tests in this class, I was able to pull that part of the test into the `setUp()` class. See Listing 9.

As with the framework's database model classes, class `ReportError` gets instantiated during `getInstance()`. Exception reporting consists of a call to its `logError()` method followed by a call to its `flush()` method. Each method should be called exactly once (`once()`).

On the other hand, each test that does expect to find the relevant record or record should not be calling either `logError()` or `flush()`. Those tests use a setup as in Listing 10.

With this setup, every test in that class verifies that `logError()` was `never()` called, and that `flush()` was `never()` called.

The `never()` tests are there for future feature development. They act as a vise[14] locking the intended functionality in place. If the error path is accidentally followed, our test suite will be complaining next time it's run.

Listing 9.

```
1. protected function setUp(): void
2. {
3.     parent::setUp();
4.
5.     $instance = FixtureReporter::getInstance($this->userRole);
6.     /** @var ReportError|\Mockery\LegacyMockInterface|\Mockery\MockInterface $mock */
7.     $mock = Mockery::mock(ReportError::class)->makePartial();
8.     $mock->shouldReceive('logError')->once();
9.     $mock->shouldReceive('flush')->once();
10.    $instance->setReporterMock($mock);
11. }
```

Listing 10.

```
1. protected function setUp(): void
2. {
3.     parent::setUp();
4.
5.     $instance = FixtureReporter::getInstance($this->userRole);
6.     /** @var ReportError|\Mockery\LegacyMockInterface|\Mockery\MockInterface $mock */
7.     $mock = Mockery::mock(ReportError::class)->makePartial();
8.     $mock->shouldReceive('logError')->never();
9.     $mock->shouldReceive('flush')->never();
10.    $instance->setReporterMock($mock);
11. }
```

Summary

I generally don't write automated unit-test suites covering trivial database "finder" or "updater" methods. However, those unit tests can take on crucial importance when the class becomes more complex. When we have business-rule processing intimately related to database access, we need to ensure future development does not accidentally break current production usage.

We took a close look at class `RoleBasedLookup` from "When You Know the Pattern." We also took a close look at the advice, and Design Patterns, related to our situation.

My own experience tells me that it's quite difficult, sometimes overwhelmingly difficult, to write clear, expressive, concise tests when the production code (the "system under test" or SUT) is working with the database. We took a careful look at known solutions to this problem.

We then walked through the current test suite for "role-based lookups." We used both test doubles and mock objects. We kept the tests disconnected from the database (that is, the tests do not depend on any actual database or database connection) while remaining relatively clear and expressive.

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

14 *vise:* https://www.merriam-webster.com/dictionary/vise

New and Noteworthy

PHP Releases

PHP 8.1.3 Released!:

https://www.php.net/archive/2022.php#2022-02-17-3

PHP 8.0.16 Released!:

https://www.php.net/archive/2022.php#2022-02-17-2

PHP 7.4.28 Released!:

https://www.php.net/archive/2022.php#2022-02-17-1

News

Symfony 6.1 will require PHP 8.1

Symfony announces that their next release will require PHP 8.1

https://symfony.com/blog/symfony-6-1-will-require-php-8-1

PHP Project Adds GitHub Actions for CI/CD

A post about the PHP Project adding GitHub actions for Continuous Integration / Continuous Delivery (CI/CD) platforms to test the PHP source code.

https://stitcher.io/blog/dealing-with-dependencies

Last talks and speakers announced for SymfonyLive Paris 2022

SymfonyLive Paris, the French Symfony conference welcomes the last conference speakers and their talks. Keynotes will be soon announced.

https://phpa.me/symfony-paris-2022

Laravel 9.2 Released

That latest version of the Laravel Framework in released and adds a `keyBy` method along with some other enhancements.

https://laravel-news.com/laravel-9-2-0

ZSH Plugin for Laravel Developers

For all you ZSH command-line commandos who happen to be Laravel developers, Jess Archer released a great new plugin for the `artisan` command.

https://github.com/jessarcher/zsh-artisan

Laracon Online Winter `22

They online version of the conference around the Laravel Framework, Laracon, took place in Feb and is freely available. Check out all the great talks on their Youtube channel

https://www.youtube.com/watch?v=0Rq-yHAwYjQ

Switch to a new branch with unstaged changes

A blog post that talks about a new feature added to Git v2.23 called `switch` which is an alternative to the classic `checkout` command.

https://phpa.me/laravel-branch-unstaged-changes

PSRs - Improving the Developer Experience

Frank Wallen

PHP-FIG (PHP Framework Interoperability Group) is the group of developers and projects coming together to define/suggest and maintain PSRs (PHP Standard Recommendations) for how frameworks and components communicate and work with each other. In PSR Alley, we will talk about the PSRs and how they are applied in, or affect, the real world of developing with PHP. The first ones we'll look at are PSR-0: Autoloading Standard, PSR-4: Autoloading, and PSR-1: Basic Coding Standard.

It is important to note that PHP-FIG membership is not limited to frameworks—it includes PHP-based projects and people notable to the industry. According to the PHP-FIG Bylaws, a member project "must be released projects with known production deployments, not aspirational projects, …" and cannot be solely plugins or extensions of existing projects.

Defining behaviors and responsibilities of systems will enable developers to swap them out for others that behave the same and communicate in the same format or to implement libraries and packages into one's codebase quickly. A PHP Standard Recommendation (PSR) is only a recommendation; it is not a requirement, developers do not have to follow them.

The value of PSRs is impossible to ignore. Having clear standards when working with other developers improves our workflows and avoids wasting time hashing out interfaces repeatedly. It fosters solid architecture and planning. Most importantly, this creates a positive workflow and communication within the communities of developers contributing to projects and frameworks. PSRs are not necessarily static, either. They change and evolve with our industry and the demands and evolution of web technologies.

The PHP-FIG maintains a clean, informative website[1], where you can read about who is involved, their membership positions, current and past PSRs, a very helpful FAQ, and the bylaws by which they operate. They also maintain a GitHub repo[2] to support the PSR packages[3].

PSR-1 describes how we should compose our code to maintain readability and expectation rather than defining how systems should interact, PSR-0 and PSR-4 describe how we can use autoloading with a standard definition of mapping our code assets utilizing namespacing. Having a common understanding of how code files are located in a codebase significantly improves the experience of other developers learning and understanding the architecture of that codebase.

PSR-0: Autoloading Standard and PSR-4: Autoloading

PSR-0 was defined to improve and ease the experience of implementing or using an autoloader. PHP's autoloader is called when an undefined class or interface is referenced. The registered autoloader function is called to locate the expected resource and `include` or `require` it. The beauty of the autoloader is avoiding a long list of includes and requires at the beginning of each script to ensure the necessary assets are loaded into memory. The conventions and standards presented by PSR-0 were based on Horde/PEAR conventions, where PEAR installed files into a main directory. By 2014, PSR-0 was deprecated in favor of PSR-4, evolving with the introduction of PHP 5.3 and beyond, which gave us proper namespacing. PSR-4 defines the specification for how a namespace should be constructed and the expectation of where the file defining the class, interface, or trait is located. It also states that an autoloader implementation:

> *MUST NOT throw exceptions, MUST NOT raise errors of any level, and SHOULD NOT return a value.*

The responsibility of the autoloader is simply to locate and load the requested asset.

The primary function of a namespace is to encapsulate or organize classes, interfaces, traits, etc. For example, one may have several classes that fall under a 'user' category: administrator and guest. The namespace could be constructed as `My\User\Administrator` in the `Administrator.php` script and `My\User\Guest` in the `Guest.php` script. The autoloader could then recognize the namespace and know where to find the source script. PSR-4 declares there must be a base directory where the namespace separators represent directory separators. Currently, it is very common to store third-party source files in a 'vendor' directory, though it could also be named something else, and the autoloader will need to know that.

1 *website: https://www.php-fig.org*

2 *GitHub repo: https://github.com/php-fig*

3 *PSR packages: https://packagist.com/packages/psr*

PSR-4 says a class with a namespace `My\User\Administrator` means the autoloader should be looking for a file named `Administrator.php` in the directory structure: `vendor\My\User`. The resulting filepath, then, resolves to `./vendor/My/User/Administrator.php`. Though deprecated, PSR-0 is still supported by PSR-4, so that the namespace `My\Data\Repositories\User_Repository` can locate the source file at a filepath like `./my-data-repositories/User_Repository.php`.

Psr-1 Basic Coding Standard

PSR-1 is simple and clear:

- Files MUST use only `<?php` and `<?=` opening tags.
- Files MUST use only UTF-8 without BOM (byte order marking) for PHP code.
- Files SHOULD either declare symbols (classes, functions, constants, etc.) or cause side-effects (e.g., generate output, change .ini settings, etc.) but SHOULD NOT do both.
- Namespaces and classes MUST follow an "autoloading" PSR: [PSR-0, PSR-4].
- Class names MUST be declared in `StudlyCaps` (later referenced as `PascalCase` in PSR-12).
- Class constants MUST be declared in all upper case with underscore separators.
- Method names MUST be declared in `camelCase`.

Some of these rules are also intended to avoid some of the real-world problems developers face. A BOM, for example, makes it easier to identify a file encoding. Still, when it's in a PHP file, this can cause headers to be sent out prematurely, resulting in an error many of us have seen: `PHP Warning: Cannot modify header information`. That can mean headers your code is expecting to emit will be ignored or cause more unexpected behavior. The rule about not mixing declaration and execution is very important to follow since executing an `include` or `require` on a file that unexpectedly makes changes (referred to as side effects) will result in unwanted behavior. Similar to a BOM, if the included/required file generates any output (echo, print, etc.), this could result in headers being sent before you want them. Worse, an `ini_set` could severely impact the application environment.

An example of mixing side-effects and declarations:

```
<?PHP

ini_set('error_reporting', E_ALL); //generates side-effects
include "a_class.php"; //generates side-effects

function foo($bar) {
    echo $bar;
}
```

According to PSR-1, we should not define the function in the same file as the `ini_set` and `include`. In the same respect, if one expects only a class to be declared in `a_class.php`, but that file contains the following line. The line has now replaced the setting from the first line of the file `a_class.php`.

```
ini_set('error_reporting', E_ALL & ~E_NOTICE);
```

The rest of the rules primarily define standards meant to support and improve workflow and communication. When reading through another developer's code, it might be frustrating and distracting (sometimes referred to as cognitive friction) to see a class name in `camelCase` when one expects them to be `PascalCase`. This cognitive friction could easily slow down code reviews or another developer coming in to help improve the code as they must continually remind themselves of the different naming conventions.

Conclusion

I hope you have a better understanding of PSR's and the community involved in maintaining them. Becoming familiar with them will make your code easier for other developers to understand, which is always appreciated when wading into a legacy codebase or building new projects.

Next Up

Next month I'll be going over PSR-12: Extended Coding Style Standard. As implied in its name, PSR-12 expands on PSR-1. It also introduces a significant set of rules in terms of number and implications.

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

Related Reading

- *PSR-7 HTTP Messages in the Wild* by Hannes Van De Vreken, April 2017. https://www.phparch.com/magazine/2017-2/july/
- *The Middleware Awakens* by Ian Littman, June 2016. https://www.phparch.com/magazine/2016-2/august/
- *Getting Started with Zend Expressive* by Chris Tankersley, March 2016. https://www.phparch.com/magazine/2016-2/march/

Finding Prime Factors

Oscar Merida

We're building on a previous puzzle for finding integer factors. In this article, we look at how to find prime factors before turning to one more puzzle involving integer division.

Prime Factors

Prime factors are critical to using modern-day encryption to secure transmitted messages. They underpin the complex maths behind encoding and decoding a message and making it computationally infeasible to decode. Another application is when working with fractions and ratios. We can use prime factors to find the lowest common multiple or the highest common factors of large numbers. Primes are also used in algorithms for random number generation to avoid repetitive patterns in the pseudorandom output. They can also be used in hashing algorithms and hash tables to minimize collisions.

Recap

Since we can find the integer factors of a number, let's find all the prime factors. Then we can check if a number is prime or not. Our puzzle last month was to write a script that will list all the prime factors of an integer. We must also say if the integer itself is prime.

For example, if the number is 24, the code should list the prime factors as:

```
2 x 2 x 2 x 3
```

or

```
2^3 * 3
```

Listing 1.

```
1. <?php
2.
3. function findFactors(int $product) : array
4. {
5.     $factors = range(1, (int) sqrt($product));
6.     $factors = array_map(
7.         function($factor) use ($product) {
8.             if ($product % $factor === 0) {
9.                 return [$factor, $product / $factor];
10.            }
11.        }, $factors
12.    );
13.
14.    return array_filter($factors);
15. }
```

If you haven't done last month's puzzle, I suggest solving that first. My solution will build on it.

Re-using Findfactors

I suspect we can re-use last month's function (Listing 1), which gives us the pairs of factors for an integer.

If we want to find the prime factors of 24, we can start with its integer factors by calling `findFactors(24)`. Doing so gives us a two-dimensional array.

```
[0] => Array
    (
        [0] => 1
        [1] => 24
    )
[1] => Array
    (
        [0] => 2
        [1] => 12
    )
[2] => Array
    (
        [0] => 3
        [1] => 8
    )
[3] => Array
    (
        [0] => 4
        [1] => 6
    )
```

We can start with one of these pairs and find its integer factors. It's probably not good to start with `1 x 24` since that restates the initial problem. But we could start with the second case `2 x 12`. We know `2` is prime, so we can add it to our list of prime factors and then get the factors of 12 and do the same. This looks like a loop (and maybe recursion, we'll come back to that…).

Recall I said, "we know `2` is prime", but there could be an infinite number of prime factors. How can we test if a number is prime? A number is prime if its only factors are one and itself. To put it in terms of our `findFactors()` function, a number is prime if the function returns one and only one pair of factors. Let's create a function `isPrime()` to test that.

Listing 2.

```
<?php

function isPrime(int $number) : bool
{
    $factors = findFactors($number);

    if (count($factors) === 1) {
        return true;
    }

    return false;
}
```

`isPrime()` seems like a handy function to have. And it's the second part of our challenge. Let's test it with some numbers:

```
var_dump(isPrime(3)); // true
var_dump(isPrime(5)); // true
var_dump(isPrime(12)); // false
var_dump(isPrime(17)); // true
var_dump(isPrime(81)); // false
var_dump(isPrime(83)); // true
```

Now, we're getting somewhere. We can take our initial number, get the first pair of factors after the `1 x` pair, and decompose its factors. I landed on a recursive solution on my first pass. After some thought, I figured it was not worth the effort to try to write this using only a loop. We should be careful. Recursion is useful when it works, but you can run into PHP's recursion limit. See Listing 3.

Listing 3.

```
<?php

/**
 * @return array<int>
 */
function findPrimeFactors(int $number) : array
{
    // could check the count here but calling isPrime
    // for readability
    if (isPrime($number)) {
        return [$number];
    }

    // Get and work with the second pair of factors
    $allFactors = findFactors($number);
    $pair = $allFactors[1];
    $keep = [];

    foreach ($pair as $factor) {
        if (isPrime($factor)) {
            $keep[] = $factor;
            continue;
        }

        $keep = array_merge($keep,findPrimeFactors($factor));
    }

    return $keep;
}
```

If we call `findPrimeFactors(24)` we get an array that looks like

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

Formatting the Output

We can output the factors in the first format with PHP's `join()` function[1].

```
echo join(' x ', findPrimeFactors(24));
```

That code produces the output shown below:

```
2 x 2 x 2 x 3
```

Producing the other desired output format might be tricky. We want to group the same factor into one bucket and get a count of each occurrence. Luckily for us, PHP has the `array_count_values()`[2] function. With it, we can output the prime factors with exponents:

```
$primes = array_count_values(findPrimeFactors(24));
$exp = [];
foreach ($primes as $factor => $power) {
    $exp[] = "{$factor}"
           . ($power > 1 ? "^{$power}" : "");
}
echo implode(' x ', $exp) . PHP_EOL;
```

Giving us the following output:

```
2^3 x 3
```

Making Change

Let's tackle one more tricky bit of division. You're given a non-zero amount of dollars. Write a script that outputs the coins and dollar bill denominations that total the amount. The program must minimize the number of bills and coins used. For this exercise, assume you can use $100, $50, $20, $10, $5, and $1 dollar bills and quarters, nickels, dimes, and pennies.

1 *join()` function:* http://php.net/join

2 `array_count_values()`: https://php.net/array_count_values

For example, if the amount is $267.51, your script should output:

```
2 $100 bill(s)
1 $50 bill(s)
1 $10 bill(s)
1 $5 bill(s)
2 $1 bill(s)
2 quarter(s)
1 penny(ies)
```

Some Guidelines And Tips

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

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

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

Related Reading

- *PHP Puzzles: Finding Integer Factors* by Oscar Merida, February 2022. http://phpa.me/2022-02-puzzle
- *PHP Puzzles: Time Value of Money* by Oscar Merida, October 2021. http://phpa.me/puzzle-oct-2021

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

I Just Can't

Beth Tucker Long

ding - Subject line: Reminder to complete your student's school registration. A new email arrives. *Right, I need to get that done as soon as I finish work for the day.* **ding - Subject line: Your domain names expire in 3 days.** *Ack, I forgot to renew those last week. I need to get that done before they expire.* **ding - Subject line: Did you get my email from last week?** *What email?* (scroll, scroll, scroll) *Oh, there it is. Not sure how I missed that…*

It starts to creep in, slowly at first—just a passing thought.

> *Oh yeah, Ticket #323 has been on the list for a while. I need to get that off of the list, but it's not a high priority, so I have to do this one first.*

A few days pass.

ding - Subject line: Have you had a chance to look at Ticket #323?

You can't help but internally roll your eyes.

> *I thought you said Ticket #323 wasn't an emergency, so why are you nagging me about it.*

But externally, at least, you manage to say, "Yes, Ticket #323 is on my list, and I will get to it as soon as I finish projects *x* and *y*."

ding - Subject line: Help! Our site was just hacked!

> *Ugh, hackers who break things on Fridays are the worst.*

Evening plans are delayed and then canceled. You finally finish mid-Saturday. Your eyes are blurry, and your head hurts, but you have to get to the post office before it closes. You are just going to wait a minute to make sure everything really is…working…correctly…

ding - Subject line: Email doesn't seem to be working

> *Oh no, I fell asleep, and now the post office is closed. I'll have to go Monday morning before standup. Email isn't working? I wonder if we are getting blocked because of spam sent out during the hack last night.*

ding - Subject line: Second Reminder to complete your student's school registration

> *Right, I still need to do that, but I have to check on the email issue before I do.*

At standup on Monday, you apologize for being late because the line at the post office was really long. You try to get out of being assigned anything new because you have a bunch of old tickets you really need to clean out, but since none of those are priority tickets, you are assigned a "quick" task to do first to set up a coupon for the marketing department.

ding - Subject line: Any updates on Ticket #323?

> *Oh no, another email about Ticket #323, which I haven't had a chance to do. I'll get it done as soon as I finish what I'm in the middle of, and then I can tell them it's done when I write back.*

ding - Subject line: Need an update on project x from you ASAP

You look back through the tickets for project x and put together a recap of where things are which you present at the meeting.

> *Ok, I'm back from the meeting, and I can finally get some work done.*

ding - Subject line: IMPORTANT - SALES CAMPAIGN SENT OUT BUT COUPON CODE NOT WORKING

> *WHAT?! Why would they send it out before I marked the ticket as done? I haven't even started it yet!*

You scramble to get the details of what this coupon code is supposed to do and frantically try to get it done as quickly as possible. At the same time, fielding questions and bug reports from each person in the marketing department and help desk individually telling you that the coupon code is not working. Meanwhile, Ticket #323 is long forgotten for yet another day.

And so it goes. There are tons of internet memes and commentaries making fun of people who can't get simple little things done. A lot of it is targeted at Millennials who can't "adult," but the truth of the matter is that this is not

laziness, a lack of intelligence, or an inability to prioritize the important tasks. This is burnout.

We are constantly inundated with communications. Finding quiet, uninterrupted time to complete tasks is akin to a quest for the Holy Grail at this point. When things are constantly interrupted, it becomes so difficult to complete even small tasks on time, and once we hit burnout, completing simple tasks becomes overwhelming.

There is no simple cure for burnout. It takes time to heal, sometimes a very long time. You can avoid it, though. Take a moment and evaluate how things are currently affecting you:

- Are you avoiding things you used to take care of easily?
- Is it becoming more common for you to get interrupted and never get back to finish your original task?
- Are you forgetting things that you normally have no trouble remembering to do?
- Are you having trouble sleeping at night? Do you fall asleep during times you don't normally sleep?
- Do you feel physically exhausted the majority of the time?
- Have your eating habits changed, eating way more or way less than normal?

These can all be signs of burnout. So what can you do? First and foremost, contact your doctor and mental health care provider. If you do not have a mental health care provider, talk to your doctor about the best options for you.

Next, take a look at what is going on in your life? Is there a way to eliminate anything causing you anxiety or stress? Are there places where you can step back or decrease your involvement (even temporarily) to give yourself some time to do something relaxing and enjoyable (and no, that does not mean time to get caught up at work - it means a hobby or something done just for fun)?

This month can be rough for many people. While sources disagree widely on what month is the toughest on mental health, February is when most people give up on their New Year's Resolutions and is considered a major low point for those who suffer from Seasonal Affective Disorder. With the recent surge in COVID-19 cases, supply chain issues, and employees buried under extra work due to staffing shortages, watching for burnout is critical.

Want to help? Now would be a great time to pass around a burnout quiz at work or host a learning lunch on identifying burnout. Remind everyone to keep an eye out for signs of burnout at the end of your next meeting. Even just a simple reminder in Slack that burnout is a real thing can help. Keep an eye on yourself as well as your friends and co-workers. Mental health is important, and so are you.

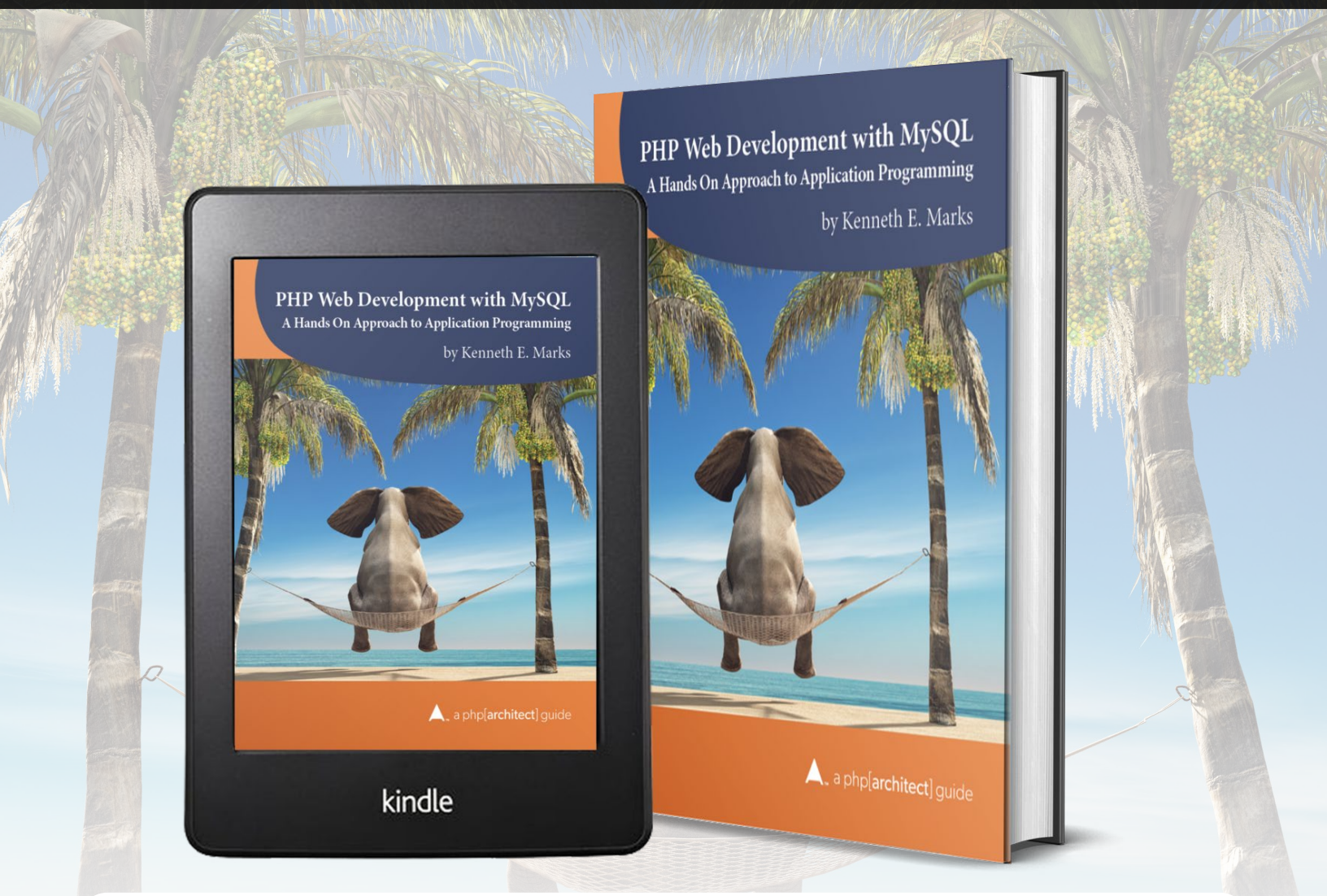
Related URLs

- *Resources*
  from Open Sourcing Mental Illness (OSMI)
  https://osmihelp.org/resources
- *Your Body Knows You're Burned Out*
  from The New York Times
  https://phpa.me/nytimes-burnout
- *Workplace Stress Survey*
  from The American Institute of Stress
  https://www.stress.org/workplace-stress-survey-pdf
- *Burnout Test*
  from Psychology Today
  https://phpa.me/psychologytoday-burnout-test
- *Burnout Quiz*
  from The Wall Street Journal
  https://phpa.me/wsj-burnout-quiz

Beth Tucker Long is a developer and owner at Treeline Design[1], a web development company, and runs Exploricon[2], a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development[3] and Full Stack Madison[4] user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @e3BethT

1 *Treeline Design: http://www.treelinedesign.com*
2 *Exploricon: http://www.exploricon.com*
3 *Madison Web Design & Development: http://madwebdev.com*
4 *Full Stack Madison: http://www.fullstackmadison.com*

a php[architect] guide



Learn how to build dynamic and secure websites.

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

Available in Print+Digital and Digital Editions.

Purchase Your Copy

https://phpa.me/php-development-book

Web Apps • Mobile Apps • E-Commerce

A DIFFERENT DEVELOPMENT EXPERIENCE

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

DIEGODEV GROUP

DIEGODEV.COM

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