www.phparch.com

January 2020
Volume 19 - Issue 1

New Habits

Object-Oriented Programming: A Primer, Part Two

Never* Use Arrays

Dealing With Devpression
Or: How I Learned to Dislike Myself Less, Part Two

Anatomy of a Browser

Sample

ALSO INSIDE

Education Station: Unit Testing Basics

Community Corner: AruzinaPHP

Pragmatic PHP: Mastering the Craft

Security Corner: Seven Deadly Sins of Security

The Workshop: Ansible in Practice

finally{}: Certainly Certifications

On Sale!

Save $200

We are the longest-running PHP developer conference in the US. This year broken into three tracks on Tech Leadership, PHP Development, and Web Technologies.

Sample

May 18-21, 2020

Nashville, TN — Opryland

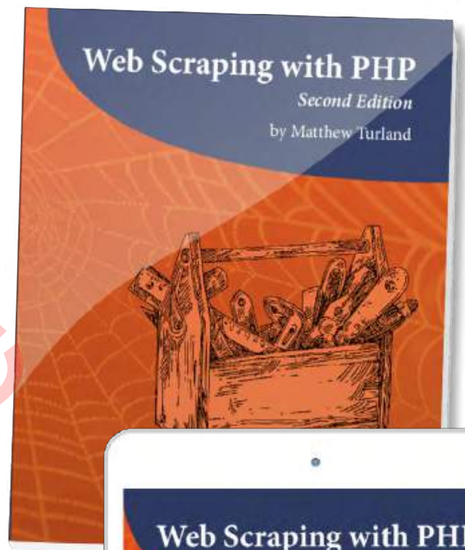
tek.phparch.com

a php[architect] guide

Read a Sample Online

Integrating with another web site but an API is not available?

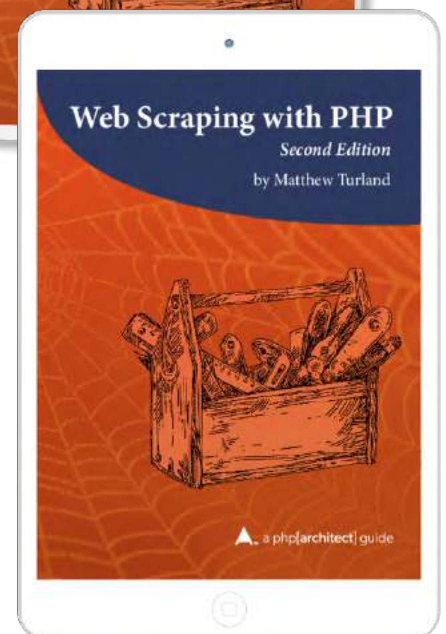
Web scraping is a time-honored technique for collecting the information you need from a web page. In this book, you'll learn the various tools and libraries available in PHP to **retrieve, parse, and extract data** from HTML.



Web Scraping with PHP, 2nd Edition includes updates to the techniques of the first edition to account for modern PHP 7 based libraries written to more easily interact with web markup and data.

- HTTP requests and responses
- PHP's HTTP Stream wrapper
- Using the cURL extension
- Working with the pecl_http extension
- Parsing responses with Guzzle
- Zend Framework's HTTP classes
- An overview of Symfony's libraries for web automation
- Writing a client from scratch
- Extensions for parsing and tidying XML and HTML
- Using regular expressions
- … and more.

Sample



Written by PHP professional Matthew Turland, this book builds on his expertise in creating custom web clients.

Available in Print, PDF, EPUB, and Mobi.

Order Your Copy
http://phpa.me/web-scraping-2ed

Unit Testing Basics

Chris Tankersley

Recently, we've discussed principles for writing clean code in your php applications. Testing is a valuable technique to help you produce and maintain a codebase, but it can be daunting to learn. In this article, we'll start with unit testing. Let's look at how tests help during the design phase and in maintenance, what unit tests are, and how to use PHPUnit for your test suite.

Over the last year, one thread I have tried to keep is the idea of writing clean, simple code. I have covered topics like dependency injection over the last few months, and before that, I talked about several concepts like DRY, SOLID, and why I feel strongly-typed code is a good idea.

The reason for this is, ultimately, new code becomes legacy code. I was recently talking with some colleagues, and I mentioned code becomes legacy the second we write it. Why? Once it's written, code becomes something the unfortunate souls that come after us have to maintain.

There is a persistent joke that "Past me was a horrible developer!" We are not bad developers. No, it's just that code decays quicker than we think.

The first line of defense against decaying code is to write simple, clean code. The harder it is to decipher the intention of code, the more difficult it is to maintain. Remember when "past me" tried to be clever and reduce that set of nested ifs into `switch()` and ternary statements? That might have worked, but it is not clear.

After clean code, the next phase is testing. The only way to make sure the code continues to work and does not break is to make sure it is easily testable. There are a few different accepted ways of how to write and run tests, each with their purposes and reasons for how they work.

Listing 1

```
1. <?php
2. // Slugifier.php
3. namespace MyApp;
4.
5. class Slugifier
6. {
7.     public function convert(string $text): string {
8.         $text = strtolower($text);
9.         preg_match('/[a-z0-9 ]+/i', $text, $output);
10.        $text = str_replace(' ', '-', $output[0]);
11.        return $text;
12.    }
13. }
```

- Unit Tests: Testing code in isolation of the rest of the codebase.
- Integration Testing: Testing code to make sure it works as expected with other units.
- Functional Testing: Testing more substantial portions of related systems at once.

Some of these may have further specializations, like behavioral testing generally borne out of a need for functional testing. Each type of test is essential for different reasons, and the idea is as you move from one to the other, you cover multiple facets of an application from various angles.

Unit Testing

Let's begin by focusing on unit testing. Unit testing is useful for a few different things. The first is testing the written code. The second is it can help determine whether or not the code makes sense before we use the code in many different places. I routinely find poor design choices in my classes using unit tests because if it is hard to write a test, it is hard to use in real situations.

Unit testing is the smallest kind of testing one can do. The basic idea of a unit test is to take a small block of code, like a method in a class, and test it. A unit test ensures the general logic works under a variety of inputs, irrespective of additional real dependencies like a database.

Consider the block of code in Listing 1, where we have a class that turns a string into a URL-friendly slug.

If I pass in a string to the `convert` method, I expect to get a new string that is all lowercase, with spaces replaced by hyphens, and all punctuation removed. How can we test this?

The easiest way to test this is to create a file that can run the tests:

```
// Slugifier.test.php
require_once 'Slugifier.php';

$slugifier = new \MyApp\Slugifier();
if ($slugifier->convert("This was some text!")
    !== "this-was-some-text") {
    throw \Exception('Text does not match!');
}
```

Sample

Anyone can run this script to make sure the `Slugifier::convert()` method is working as expected. We can now change the internal implementation of the `convert()` method whenever we need to and make sure it does not break. We have a small test that makes sure `convert()` does what we expect it to, and nothing more or less.

We can then build onto this script and build more test scripts for more of our codebase.

Is any of this super elegant? No, but now anyone can run the test with `php Slugifier.test.php`, and that is better than nothing. From here, you could write a test runner which searches through your source code for `*.test.php` files and can execute the files, and stop when it encounters an error.

> *This is very close to what the PHP Test Runner does for the core PHP language itself. The big difference would be that PHP tests (usually denoted by a `.phpt` extension) test output directly, and not necessarily failure conditions.*

PHPUnit

There is a better and more accepted way to write tests. The very basic way I show above works, but it fails at a few things. First, unit tests should run in total isolation. The code should be executed from a clean slate every time in addition to testing small chunks of code. The above setup makes it very easy to break this. The non-elegant solution also has no way to deal appropriately with dependencies.

Fortunately, there is an excellent tool that has been around since 2001, PHPUnit[1]. PHPUnit uses the xUnit architecture, in turn derived from Smalltalk's SUnit. xUnit describes how to group and execute tests and how to deliver the results.

Installing PHPUnit

Installing PHPUnit is as simple as any other library using Composer, but it should be declared as a development dependency and not as a normal dependency. This way, PHPUnit is only installed when doing development and can be ignored when installing dependencies for Production. We can use Composer like normal and pass an additional `--dev` flag:

```
composer require --dev phpunit/phpunit ^8.5
```

Do not install PHPUnit on your production machines. I'm going to tell you the same thing the PHPUnit documentation says:

> *If you upload PHPUnit to a webserver then your deployment process is broken. On a more general note, if your `vendor` directory is publicly accessible on your webserver then your deployment process is also broken.*

Writing Tests

Writing tests with PHPUnit is straightforward and uses various naming conventions for how to set things up. You define a class which extends `PHPUnit\Framework\TestCase`, name it `[Class]Test`, and create public methods starting with the word `test`. It has some built-in assertions, or methods, for testing the validity of data, that we can call to make sure code works correctly.

Where do we put this code? PHPUnit expects all of the tests to live in their folder. In PHP, the typical convention is to have a `tests/` directory in the root of your project, and under there, follow a similar folder structure to the other source code. The actual test files are `[Class]Test.php`.

```
|____src
| |____MyApp
| | |____Slugifier.php
|____composer.json
|____tests
| |____MyApp
| | |____SlugifierTest.php
|____composer.lock
```

We can convert the above test without much work (see Listing 2).

This test looks broadly the same, but the difference is our test is now wrapped in the convention PHPUnit expects. Since we now extend `TestCase`, we can use a method called `assertSame()` to test that two values are the same. We pass in argument 1 as the string we expect, and argument 2 as the result from our own `convert()` method.

PHPUnit allows quick access to various types of assertions[2], including:

- `assertSame(mixed $expected, mixed $actual)`: Make sure the values of two things match.

Listing 2

```
1. <?php
2. // tests/MyApp/SlugifierTest.php
3.
4. use PHPUnit\Framework\TestCase;
5. use MyApp\Slugifier;
6.
7. class SlugifierTest extends TestCase
8. {
9.    public function testConvertWorksProperly() {
10.      $slugifier = new Slugifier();
11.
12.      $this->assertSame(
13.         "this-was-some-text",
14.         $slugifier->convert("This was some text!")
15.      );
16.   }
17. }
```

1 PHPUnit: https://phpunit.de

2 various types of assertions: https://phpa.me/phpunit-8-5-assertions

Sample

- `assertEqual(mixed $expected, mixed $actual)`: Make sure two things are the same, like that two objects are the same instance.
- `assertTrue(bool $condition)` and `assertFalse(bool $condition)`: Make sure values are true or false.
- `assertArrayHasKey(mixed $key, array $array)`: Make sure an array key exists.
- `assertCount(int $expected, $haystack)`: Make sure the `$haystack` has the correct count.

If we have `composer.json` set up to autoload our `MyApp` namespace, we can run the PHPUnit test runner and point it to `vendor/autoload` as a bootstrapping script, and then to the `tests/` directory to run our tests.

```
vendor/bin/phpunit --bootstrap vendor/autoload.php tests/
```

PHPUnit searches through the `tests/` directory for PHP files to execute, and any tests it finds.

Anatomy of Good Tests

Writing a test is one thing, but writing a good test is something else. One needs to resist the urge to throw a bunch of stuff inside of a method and call it a day—a good unit test has a few components and ideals that make a test worthwhile. What you do not want to end up with is a bunch of tests that just do the bare minimum or don't really test anything at all.

The first idea I try and stick to is that each test tests one idea, and preferably should only execute one operation. A test should not test both retrieving data and saving it; those should be two separate tests. Listing 3 is an example of a poor test.

This test's name says it is testing whether the `convert()` method works. In reality, this test is hitting a database, finding a specific post, making sure it is an object, then using that `$post` object as part of the test. Doing all of those operations is beyond the intended scope of a unit test. We should be testing only one operation—in this case, that a string is turned into another string. Anything more is too much work for the test. We should split this single test into at least two total tests:

1. one for the `Slugifier` service,
2. and the other for the `PostService::find()` method.

Each of these is a separate unit of work. By having independent tests to verify each runs as intended, you can be confident both should work when used together. And if something fails, having two tests makes it easier to identify which component was at fault.

This does not mean you cannot have multiple assertions per test. You should have one logical assertion per test, but that can be broken down into multiple assertions. What do I mean? We could, for example, test whether our fictional `PostService` class returns a `Post` object. That is our "logical" assertion. To make sure the `Post` object is correct, we can run multiple, smaller assertions that help us reach our "logical" assertion (Listing 4).

The difference between these ideas is the number of logical assertions. In the bad test, we did multiple operations before we could run the tests, and even partially tested the `Post` object. In the second test, we did one operation, finding a post, and then made sure the post looked correct. You can have multiple assertions per test, but limit the number of operations per test.

The second thing to think about is not only successful operations but also failure operations. For our `PostService::find()`

Sample

Listing 3

```
1. <?php
2.
3. use PHPUnit\Framework\TestCase;
4. use MyApp\Slugifier;
5. use MyApp\PostService;
6.
7. class SlugifierTest extends TestCase
8. {
9.    public function testConvertWorksProperly() {
10.       $slugifier = new Slugifier();
11.       $posts = new PostService();
12.
13.       $post = $posts->find(1);
14.
15.       $this->assertIsObject($post);
16.
17.       $this->assertSame(
18.          $post->getSlug(),
19.          $slugifier->convert($post->getTitle())
20.       );
21.    }
22. }
```

Listing 4

```
1. <?php
2.
3. use PHPUnit\Framework\TestCase;
4. use MyApp\PostService;
5. use MyApp\Post;
6.
7. class PostServiceTest extends TestCase
8. {
9.    public function testPostLooksOKWhenRetrievedWithFind() {
10.       $id = 1;
11.       $service = new PostService();
12.       $post = $service->find($id);
13.
14.       $this->assertIsObject($post);
15.       $this->assertInstanceOf(Post::class, $post);
16.       $this->assertEqual($id, $post->getId());
17.       $this->assertEqual('Why Testing Is Good', $post->getTitle());
18.    }
19. }
```

method, we made sure we could find a post. What happens when we cannot find a post? Should we test that? Yes, we should! If we expect an exception is thrown, we can use `$this->expectException()` to flag that we expect our test to throw an exception, and that we expect it to be a specific type of exception, see Listing 5.

Third, a short unit test is more readable and preferred over a long unit test. If you keep to one operation per test, this should drastically help cut down on the amount of stuff inside a single test. A developer should be able to skim the test and understand what is going on quickly. Complicated test setups mean either your tests are doing too much, or your objects are too complicated and need restructuring. Remember, each class should have one responsibility.

Listing 5

```
1. <?php
2.
3. use PHPUnit\Framework\TestCase;
4. use MyApp\PostService;
5. use MyApp\Post;
6. use MyApp\PostNotFoundException;
7.
8. class PostServiceTest extends TestCase
9. {
10.    public function testExceptionThrownWithBadPostID() {
11.       $this->expectException(PostNotFoundException::class);
12.
13.       $service = new PostService();
14.       $post = $service->find('asdf');
15.    }
16. }
```

Handling Lots of Test Data

Our test case for our `Slugifier::convert()` method is pretty good, but what happens when we want a unit test to have a lot of data? PHPUnit has what it calls a "data provider," which is a function which returns an array of arrays that are passed into a test. This is especially useful when you need to test many different permutations of data, but a single test can handle all of these permutations.

You can add an annotation to the DocBlock of a test with `@dataProvider [method]`, and PHPUnit passes the return values of that method into your test (see Listing 6). The data provider should return an iterable, which yields an array or an array of arrays, with the internal array being the arguments for your test. If we augment our test to take in a `$expected` string and a `$text` string, our data provider needs to return an array of two-element arrays, with element 0 being our expected output, and element 1 being our test text to pass in.

Data providers are a great weapon against regression testing. As users and testers file bugs, they can be added to the data provider to make sure they never break again.

Listing 6

```
1. <?php
2. // tests/MyApp/SlugifierTest.php
3.
4. use PHPUnit\Framework\TestCase;
5. use MyApp\Slugifier;
6.
7. class SlugifierTest extends TestCase
8. {
9.    /**
10.    * @dataProvider sluggableStringsProvider
11.    */
12.   public function testConvertWorksProperly(string $expected,
13.                                            string $text) {
14.      $slugifier = new Slugifier();
15.
16.      $this->assertSame($expected, $slugifier->convert($text));
17.   }
18.
19.   public function sluggableStringsProvider() {
20.      return [
21.         ['this-is-text', 'This is text'],
22.         ['this-is-text', 'This is text!'],
23.         ['this-is-text', 'THIS IS TEXT'],
24.         ['hello', 'h%$^&e@#l#$l^&o'],
25.      ];
26.   }
27. }
```

Sample

Dealing with Dependencies

Our simple `Slugifier` class has no dependencies, but most code has some interaction between classes. As I have discussed in earlier articles, these are dependencies and, no matter how well you structure your code, you always have some coupling—or interaction—between different classes.

If we have an `Invoice` class (Listing 7) that requires some sort of `WriterInterface`-based dependency, which generates the actual invoice, we have a dependency.

Listing 7

```
1. <?php
2.
3. class Invoice
4. {
5.    protected $id;
6.    protected $writer;
7.
8.    public function __construct(string $id,
9.                                WriterInterface $writer) {
10.       $this->id = $id;
11.       $this->writer = $writer;
12.   }
13.
14.   // ...
15.
16.   public function generate(): string {
17.      return $this->writer->write($this->getData());
18.   }
19. }
```

Unit tests, however, test things in isolation. We should not create an actual writer object to pass in, but we should create a fake version we can control. This design allows us to test specific conditions for the test without worrying about setting up a full writer or worrying about side effects from using the actual object. The most common reason is to help control the output and behavior of the dependency to make sure all conditions, both good and bad, happen.

PHPUnit can create fake objects, which are properly called test doubles. PHPUnit can create stubs, which are test doubles that return data, or mocks, which allow some more introspection on the way we call the test double. For example, let's create a stub `WriterInterface` which returns a pre-set string as in Listing 8.

Listing 8

```php
1. <?php
2.
3. namespace MyAppTest;
4.
5. use PHPUnit\Framework\TestCase;
6. use MyApp\Invoice;
7.
8. class InvoiceTest extends TestCase
9. {
10.    public function testWrite() {
11.       $expected = 'This was output';
12.
13.       $writer = $this->createStub(\MyApp\WriterInterface::class);
14.       $writer->method('write')
15.              ->will($this->returnValue($expected));
16.
17.       $invoice = new Invoice('123-abc', $writer);
18.       $output = $invoice->generate();
19.
20.       $this->assertSame($expected, $output);
21.    }
22. }
```

We create a stub using `$this->createStub([ClassName])`, which returns an object we can start to manipulate. We then tell it to add a method called `write()` and to have it always return `"This was output"` whenever it is called. This direct control over the stub's return value is where we can see how the class and method we are testing, `Invoice:generate()`, can react to different values. You can even have the method throw an exception to test error handling.

If you need more control, PHPUnit's mock system allows you to validate the data coming into the test double. PHPUnit also has built-in support for Prophecy[3], which is a mocking framework. I use this all the time at work as our unit tests deal with PSR-7 requests; Prophecy makes it extremely easy to validate data, like a JSON body we intend to send, is coming into our test doubles to make sure we are formatting this correctly.

More information on creating stubs and mocks can be found in the PHPUnit Documentation on Test Doubles[4].

There is More

PHPUnit has a large, rich set of functionality that is impossible to capture in a single article. I have just scratched the surface of writing unit tests, and there are other types of tests for us to examine! Next month, we'll dive into functional and integration testing and where it fits in next to unit tests.

All of the above should give you a good base for writing tests. PHPUnit is an incredibly well-documented library. You can find more information about the general usage of PHPUnit in their Getting Started[5] section and their excellent documentation[6].

Sample

Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. He works for InQuest, a network security company out of Washington, DC, but lives in Northwest Ohio. 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 *Prophecy:* https://github.com/phpspec/prophecy

4 *Test Doubles:* https://phpa.me/phpunit-8-5-doubles

5 *Getting Started:* https://phpa.me/phpunit-8-started

6 *documentation:* https://phpunit.de/documentation.html

Related Reading

- *End-to-End Testing Automation With PuPHPeteer* by Gabriel Zerbib, September 2019. https://phpa.me/zerbib-puphpeteer
- *Testing Strategy With the Help of Static Analysis* by Ondrej Mirtes, April 2018. https://phpa.me/apr-18-testing-static-analysis
- *PHPUnit Worst Practices* by Victor Bolshov, April 2018. https://phpa.me/phpunit-worst-practices

Borrowed this magazine?

Get php[architect] delivered to your doorstep or digitally every month!

Each issue of php[architect] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe

Sample