PHPUnit Starter

Learn the 20% of PHPUnit You'll Be Using 80% of the Time

Elnur Abdurrakhimov

PHPUnit Starter

Learn the 20% of PHPUnit You'll Be Using 80% of the Time

Elnur Abdurrakhimov

This book is for sale at http://leanpub.com/phpunit-starter

This version was published on 2016-05-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

PHPUnit is the defacto testing framework for PHP. If you want to start automating tests of your code, PHPUnit is the right tool for the job.

Why Automate Tests

Manual testing is an error prone and ineffective process.

Testing the same things over and over can bore anyone to death. And since that's so boring, the natural tendency is to stop paying attention to details and miss bugs. Humans are not meant to do repetitive and mindless work; they're meant to enjoy fun and creative activities like programming machines to do repetitive and mindless work for them.

And the good news is that while machines are dumb, they're very fast at doing what they've been programmed to. So it makes perfect sense to teach them how to test our code and let them do that whenever we need them to.

Types of Tests

There are a lot of types of automated tests:

- Unit,
- Component,
- Integration,
- Functional,
- UI/API,
- End-to-end,
- and so on.

Even though on a project with heavily automated testing you'll meet test of most of those categories, unit tests are the easiest to start with and that's what we'll focus on in this book. After you get familiar with PHPUnit and unit tests, you'll be able to use it for other types of tests, too.

Who Is This Book For

I assume you're familiar with PHP. You don't have to be an expert; knowing basics of the language and having experience with actually writing PHP code is enough.

I don't assume you know anything about automating tests and hence I'll introduce you to whatever you need to know to write tests with PHPUnit.

How You Should Read This Book

This book is written in a tutorial style and each chapter flows into the next one. Hence you should read the book from cover to cover and not skip chapters.

Besides, the book is so short that there should be no reasons to jump ahead. The whole book can be read in a day or two.

Sample Project

Since this is not an abstract book, we'll be writing actual code. And to do that, we need a sample project to focus on.

I don't want to bother you with learning an unfamiliar domain just to be able to read this book, so the project has to be simple. But in order to have enough problems to automate tests for to show you the features of PHPUnit you're very likely to use, it has to be not too simple.

Taking all that into account, I came up with an idea to write a simple email address checker subsystem.

Imagine you're working on some kind of a social network where people sign up with an email address and a password. One of the system requirements is to not

let people sign up with an email with certain characteristics. For instance, an email ending with `@example.com` is invalid because `example.com` is a reserved domain and no one actually uses it.

You've been tasked with creating such a blacklist email address subsystem and that's what we'll be doing in this book. It won't be sophisticated enough for real use, but it will be enough for you to learn PHPUnit.

But first let's install PHPUnit and ensure it works.

Installation

The easiest way to install PHPUnit — or any other PHP library or framework for that matter — is to use Composer[1]. Let's install Composer first.

Installing Composer

If you're using Linux/Unix/OSX, the installation process is simple. All you have to do is to run the following command in a terminal:

```
$ curl -sS https://getcomposer.org/installer | php
$ mv composer.phar /usr/local/bin/composer
```

To test that Composer has been installed successfully, run the following command:

```
$ composer -V
```

If everything went well, you should see output similar to this:

```
Composer version 1.0-dev (feefd5...54d0d5) 2015-12-03 16:17:58
```

If you're getting that output, you can move on to the next section.

If you're not getting that output or using another OS, please go through the official installation instructions[2].

[1] https://getcomposer.org/

[2] https://getcomposer.org/doc/00-intro.md

Configuring the Project and Installing PHPUnit

Now that we have Composer installed, let's configure our project and install PHPUnit.

Create a directory for the sample project we're going to work throughout the book. I called it `phpunit-starter` in my system, but you can come up with any other name if you don't like that one.

Now, create a file called `composer.json` in that directory with the following contents:

```
1  {
2      "require": {
3          "phpunit/phpunit": "^4.8"
4      },
5      "autoload": {
6          "psr-4": {
7              "": "src/"
8          }
9      },
10     "config": {
11         "bin-dir": "bin"
12     }
13 }
```

This file tells Composer how to handle our application. Let's go through it.

On lines 2-4, we have a `require` section. This section tells Composer which packages our application needs — project dependencies. The only dependency we have here is PHPUnit that we specify on line 3. `^4.8` is a version constraint that tells Composer to install PHPUnit of version 4.8 and up to but not including version 5.

> I limited PHPUnit with version 4.8 here because 5.x requires PHP 5.6+ and not many people have upgraded to it yet. PHPUnit 4.8 works on PHP 5.3+. If you're worried that you're learning an older version of PHPUnit, fear not. The basics

> covered in this book work the same in both PHPUnit 4 and 5.

On lines 5-9, we configure autoloading of PHP classes. We need that to avoid including all the files we need with `include()`/`include_once()`/`require()`/`require_once()` statements. Thanks to autoloading, classes will be found and loaded automatically.

On lines 10-12, we tell Composer to copy binary files that a dependency our project depends on can ship with. PHPUnit does ship with one. We need that to get the `phpunit` binary installed into the `bin/` directory instead of the `vendor/bin`. We do that just for simplicity of executing PHPUnit.

Now that we have `composer.json`, let's tell Composer to do its job:

```
$ composer install
```

If everything went as expected, you should see output similar to this:

```
Loading composer repositories with package information
Installing dependencies (including require-dev)
  - Installing sebastian/version (1.0.6)

... omitted ...

  - Installing phpunit/phpunit (4.8.19)

... omitted ...

Generating autoload files
```

Now, several things should appear in the project directory. The `vendor/` directory contains the dependency we specified in our `composer.json` file along with dependencies of that dependency and dependencies of those dependencies — you get the idea. It also includes autoloading-related files generated by Composer.

`composer.lock` is the file listing exact versions of installed dependencies. It makes sense to commit it to your project's VCS to ensure that all team members and servers get exactly the same versions of dependencies to avoid hard-to-debug bugs caused by installing a dependency of a different version on some server. We don't really care about that file in this book.

The last but far from the least thing that appeared in the project directory is the `bin/` folder. It appeared here thanks to the configuration we did in our `composer.json`. It's the most important thing that Composer added for us because it contains the `phpunit` executable.

Now, let's verify that PHPUnit got installed correctly by running the following:

```
$ bin/phpunit --version
```

If everything went well, you should get output similar to this:

```
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.
```

That line means that we got PHPUnit installed and are ready to start learning it. It's time to write our first test.

The First Test

Remember our sample project? We need a system that prevents users from signing up with email addresses having particular characteristics. One of those characteristics is the use of a domain like `example.com`. So the first thing we need is to extract the domain part from an email address. Let's start with a test for a class that will extract parts of an email address.

Writing the Test

We're going to start with a test instead of writing the actual code and then adding tests for it. A benefit of that approach is to be able to figure out the most convenient way to use our class as a client. This leads to better code because if it's hard to use a class in tests, it's going to be hard to use for clients as well.

Another benefit of writing tests first is to define the boundaries of what our "real" code is supposed to do. If all tests pass, we're either done or have to add more tests.

We will come back to these and other benefits of writing tests first later — after you've had enough of instant gratification of writing a couple of tests and making them pass. For now, let's get to work.

We need a directory to put our tests into. Let's create a directory called `tests` in the directory of the project.

Now, let's create the file for the test class we're going to write. Let's call it `EmailAddressPartsExtractorTest.php`.

Finally, let's write the test itself:

```
1  <?php
2
3  class EmailAddressPartsExtractorTest
4      extends PHPUnit_Framework_TestCase
5  {
6      public function testExtractDomain()
7      {
8          $extractor = new EmailAddressPartsExtractor();
9          $this->assertEquals(
10             "example.com",
11             $extractor->extractDomain("elnur@example.com")
12         );
13     }
14 }
```

On line 3, we define the `EmailAddressPartsExtractorTest` test class that extends `PHPUnit_Framework_TestCase`. `PHPUnit_Framework_TestCase` is the class all PHPUnit test classes need to extend in order to be treated as such by PHPUnit. If we didn't, PHPUnit would just skip that class instead of looking for test methods in it.

On line 6, we define the `testExtractDomain()` test method. PHPUnit knows that it's a test method because of the `test` prefix. Not all methods in a test class have to be test methods. If you define a method without the `test` prefix, PHPUnit will just skip it.

On line 8, we create the `$extractor` instance of the `EmailAddressPartsExtractor` class. Don't worry that the class doesn't exist yet; we'll get to it soon.

On lines 9-12, we call the `assertEquals()` method that comes with the `PHPUnit_Framework_TestCase` class that our test class extends. We pass two arguments to it: the expected result and the actual result of calling the `extractDomain()` method of our not yet existing class. According to the name of the method, it's supposed to extract the domain part of an email address passed to it.

Let's now run the test and see how it goes:

```
$ bin/phpunit tests
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

PHP Fatal error:  Class 'EmailAddressPartsExtractor' not found in /home/elnu\
r/proj/phpunit-starter/tests/EmailAddressPartsExtractorTest.php on line 8
```

It failed as expected. Let's fix the problem by defining the missing class.

Making the Test Pass

Before we add the missing class, we need to create a directory for the "real" code of our application. Remember the `autoload` section from we added to the `composer.json` file? It points to the `src` directory of our project. Let's create that directory then.

Now that we have the directory, let's create the file that will hold the class. Let's name it `EmailAddressPartsExtractor.php` and fill it with the following code:

```
<?php

class EmailAddressPartsExtractor
{
}
```

As you can see, that's not much. But let's rerun our test again:

```
$ bin/phpunit tests
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

PHP Fatal error:  Call to undefined method EmailAddressPartsExtractor::extra\
ctDomain() in /home/elnur/proj/phpunit-starter/tests/EmailAddressPartsExtrac\
torTest.php on line 11
```

We solved the previous problem but ran into another one — the missing method. Let's add it:

```
<?php

class EmailAddressPartsExtractor
{
    public function extractDomain($emailAddress)
    {
    }
}
```

And let's rerun the test again:

```
$ bin/phpunit tests
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

F

Time: 38 ms, Memory: 4.00Mb

There was 1 failure:

1) EmailAddressPartsExtractorTest::testExtractDomain
Failed asserting that null matches expected 'example.com'.

/home/elnur/proj/phpunit-starter/tests/EmailAddressPartsExtractorTest.php:12

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

That's much better. Now, instead of PHP itself crashing with fatal errors because of an undefined class or method, we get PHP executing properly and instead see our PHPUnit assertion fail. As you can see, we get a completely different output in this case. Let's go through it.

On line 4, we see an `F` that stands for "failure". We get a single character for each test we execute. Since we only have one test now, all we see is a single character. We'll see more when we add more tests.

On line 6, PHPUnit reports how much time it took to run all tests and how much RAM was used by PHP.

Line 8 is self-explanatory.

On lines 10-11, PHPUnit tells us about the failed assertion. First it tells us which test has failed and then reports the reason it failed. That reason is specific to each assertion method. In our example, we use `assertEquals()` that expects the expected and actual values to match. Since our `extractDomain()` method doesn't have any code in it, it returns `null`. And that's what the assertion failure tells us about.

On line 13, PHPUnit reports the test class and the exact line the of the failed assertion. Since we split the assertion over 4 lines of code, it points to the last line of that `assertEquals()` method call.

On line 16, PHPUnit tells us about the total number of tests methods run, the total number of assertions executed, and the total number of failed assertions. Since a test method can have multiple assertions in it, that makes sense.

> If you want colored output — and I bet you do — pass the `--color` argument to the `phpunit` executable:
>
> ```
> $ bin/phpunit --color tests
> ```
>
> Now you'll get the 4th and 15-16th lines highlighted with red. That makes it much easier to quickly determine whether your tests failed or passed.

What's left now is to make the test pass. Let's do that now:

```php
<?php

class EmailAddressPartsExtractor
{
    public function extractDomain($emailAddress)
    {
        return 'example.com';
    }
}
```

```
$ bin/phpunit --color tests
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

.

Time: 37 ms, Memory: 3.75Mb

OK (1 test, 1 assertion)
```

Yay! It's so much nicer to see green instead of red. Our first test is passing. And the output is much shorter this time. That's because there are no failures for PHPUnit to report about.

The `F` on the line `4` got replaced with a dot. A dot means a passing test.

And the last 8th line says `OK` and reports how many test methods and assertions were executed. Since there are no failures here, PHPUnit doesn't mention them.

But wait! Something is definitely wrong with our domain extracting implementation. Instead of writing some real logic, we just hardcoded the expected result. Isn't that cheating?

Well, not really. I mean, we made the test pass, right? Who cares how exactly we did that? Oh, you do? Okay. Scroll to the next chapter and we'll deal with that there.