THE GRUMPY PROGRAMMER'S PHPUNIT COOKBOOK

CHRIS HARTJES

The Grumpy Programmer's PHPUnit Cookbook

Chris Hartjes

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

This version was published on 2014-01-21

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

Foreword

Justin Searls is what I like to call a "testing brother from another mother". He works with the awesome folks at Test Double[1] *providing consulting services for people doing Ruby and Javascript, with a focus on testing. He and I have had some interesting discussions about testing in general and I asked him to provide his perspective on the benefits of testing.*

I first used PHP to build a web application in 2004. By that time, I'd already been taught – and had failed to learn the benefits of – unit testing and even test-driven development. In this case, I just wanted to get my application written, with as few secondary concerns to worry about as possible.

Developing in PHP made me happy. Both the language and its community seemed focused squarely on helping me get things done, and they didn't impose an expectation that I adopt any highfalutin' practices or conventions. As a result, I was iterating on a working application from day one.

And "day one" was hugely productive! The same went for "week one". As it happened, "month one" was such a success that I had completed the minimum requirements of an application for which I'd been given over 20 weeks' worth of budget.

I soon discovered, however, that there was trouble in paradise. Slowly, adding small features had begun to take me significantly more time than I'd been accustomed to large features

[1] http://testdouble.com

taking. While I'd previously been a beacon of workplace optimism and excitement, I noticed that I was spending a larger proportion of my day feeling confused and frustrated. I wasn't only dreading showing up for work, I'd even begun to resent my editor's application icon, because clicking it meant that I was in for a bad time.

The root cause, and it's plainly obvious to anyone who might read my code, is that the source had grown into a tangled jungle of rogue mega-functions, replete with trees of towering switch statements and swamps of nested if-else constructs. The code lacked any rhyme or reason. It contained no helpful signposts to my future self. Concerning myself with the design of the code hadn't seemed necessary, because my intimate, almost instinctual knowledge of the code had made me so productive at the project's outset.

But at some point, the complexity of my application's code had increased beyond what I could hold in my brain at a single time. Prior to that point, I had been unfettered; liberated from formalities like up-front design or maintaining automated tests. Beyond that point, though, I could sense that my haste had cornered me into a dead end. The only question was: could I re-bottle whatever genie I'd released to help me become so productive in the first place?

In the case of that application, the answer was "no"; I couldn't make things better. The drastic internal changes necessary to make the application easier to maintain would have required confidence that my changes didn't break anything. At the time, I lacked the expertise to write the automated tests I would have needed to gain said confidence to make any major changes. Ultimately, I relented and stayed on the path of most resistance, muscling my way through to the end of the

project. Needless to say, no one was as impressed with my productivity during the final leg of the project as they had been after its glorious "month one".

At this point, it might seem like I told this cautionary tale for the purpose of exhorting to readers, "*see?! That's why you start writing tests for everything on day one! Because even if you feel great today, someday your application will explode with complexity and you'll be miserable*". I won't say that, however, because that would be a specious, dogmatic argument and it would oversimplify the world of rich and subtle challenges facing software developers. There are, after all, many ways to ship working software with acceptable maintainability; excellent software can absolutely be written without any automated tests at all.

So if testing isn't absolutely necessary for success, why is it valuable? The answer is that automated tests are an excellent tool for establishing tight, rapid feedback loops when creating and changing code. To explain, let me rewind to the beginning of my story.

I said that working with PHP made me happy, because I felt the extraordinary positive feedback of seeing something, albeit small, start working in the first couple of hours. Not only that, but I could get ongoing feedback that my changes worked as quickly as I could save a file and refresh a browser window. It was only once my application had grown significantly that the feedback loop had started to slow down – a five second pause here, a few clicks to set up the app there, perhaps a few minutes to verify the change didn't introduce any bugs. What I eventually realized that I wanted was for every day to feel like "day one" of a fresh project; I wanted a rapid feedback loop to be sustainable for the life of the application.

In the past, I had tried building applications of a similar scope with other tech stacks, like Java, known for their long-term "hardiness". But my projects never seemed to get off the ground. I failed in part because I'd sink the first, crucial hours of my motivation into troubleshooting while setting up the recommended build tools and supporting libraries. And even when I managed to clear that hurdle, any sense of progress was stymied by the nagging doubt that my architecture would elicit the judgment of my contemporaries. Perhaps a more durable technology stack or application design would have made my rapid feedback loop more sustainable in the long run, but the initial "short run" was so painful that I'd never find out what the long run" felt like.

It turns out that a "sense of progress" is crucial to productive software development. Feedback, both positive ("that worked!") and negative ("that doesn't!") has a huge impact on the psyche of the developer. When people endeavor to build tangible things, they receive concrete feedback about progress constantly (e.g. "the table has two legs, so I'm about halfway done"; "this book introduction has 283 words, so I've got a ways to go"). But when building something as ephemeral as software, progress comes in fits and starts, sometimes to the point of feeling illusory.

The magic of unit testing, particularly in the context of test-driven development, is that it gives the developer the ability to control his or her own sense of progress. Traditional feedback requires our fully integrated application and our eyeballs to assert whether we're on the right track or wrong track. Unit testing allows us to establish a feedback loop at whatever level-of-integration we wish (e.g. perhaps a bunch of objects in coordination, perhaps one function in pure isolation), so long as we can imagine and implement a way to assert

working behavior that doesn't require our eyeballs' manual inspection. In this way, even if we're faced with implementing a solution of daunting complexity, unit tests can usually help us break the problem down in such a way that we can make (and importantly, feel!) incremental progress on our path to overall completion.

By mastering both the tools and craft of unit testing, rapid feedback is attainable regardless of the age or complexity of the project. At the outset of an application's life, a failing test can help us set up the critical infrastructure of our application, and we can get some motivating feedback even if there's nothing visible to users yet. And for a mature project, however tangled its source, a test can usually be crafted to gain certainty that a change was made safely and successfully.

When applied rigorously and consistently, and when the pain of a hard-to-write test is responded to with improvement to the design of the code being tested, developers can hope to remain happy and productive over the life of any application.

There's a lot more to learn about how to get there, but that's what the rest of this book is for! My hope is that the lessons that Chris has prepared in this book will serve to equip you with the skills to someday find yourself working on a mature application while feeling as productive as you did on day one.

- Justin Searls

Acknowledgements

Books don't happen without lots of help from other people. That said, I wish to thank the following people:

- Sebastian Bergmann and other PHPUnit contributers for providing a powerful tool for testing PHP code
- Kim Rowan for her editing efforts
- Chris Tankersley and Matthew Weier O'Phinney, for their technical editing
- Members of Amy Hoy's 30x500 course for conversations about marketing and selling things to developers
- My PHP Testing Bootcamp students for reminding me about what I take for granted
- conference organizers who have graciously allowed me to promote my ideas on testing and application building
- Tanya Lam (http://www.tanyalam.com/) for the awesome cover art

A special thanks must go out to people who take the time to discuss issues involved with testing software applications with me. Your questions and thoughts inspire me to keep sharing the things I learn.

Finally, none of this would be possible without the incredibly high tolerance levels of my wife, Claire. She handles my frequent absences (both real and virtual) with grace and humour.

Introduction

When I wrote my book "The Grumpy Programmer's Guide To Building Testable Applications In PHP" my goal was to teach people how to write code that you could easily test. My reasoning was that there was lots of information available on how to use testing tools.

Turns out I was only partially right.

While using your search engine of choice can show you how to accomplish certain tasks, it was difficult to find one place that showed anything beyond extremely shallow solutions.

I did some more research. I signed up for an awesome product development course that taught me how to do even more research, and I started creating a solution that I was certain would help people solve the pain of how to actually write tests for their PHP code using PHPUnit.

The result is this book. I've tried to give you examples of code that we are trying to write tests for, along with explanations about the decisions that I've made.

Don't think of this book as something that you will read end-to-end. It's far more likely that you will end up using it a chapter at a time, learning the skills to give you a solid understanding of just one part of the testing process.

As always, I welcome your feedback via Twitter and App.net (@grmpyprogrammer) or via email at chartjes@littlehart.net.

PHPUnit For Grumpy Developers

PHPUnit can look intimidating, even to just get the skeleton of a test created, due to the immense length of its documentation and large number of configurable options. Don't be scared though, I am here to help!

You can start with just a few of the basics before moving on to more complicated setups that include options for skipping certain types of tests or changing default settings.

To be honest, the defaults will cover 99.999% of your testing needs.

Installing And Configuring

Installing PHPUnit and its associated (sometimes optional) dependencies has gotten easier and easier with each passing day. As I write this book in the Canadian winter of 2012-2013, I have two preferred ways of installing PHPUnit that I would recommend.

Installing via Composer

Composer[2] is a command-line tool for tracking and installing dependencies for your application. In my opinion, Composer

[2] http://getcomposer.org/

is transforming the way PHP developers build their PHP applications, making it easier to install dependencies and external libraries. With all the major frameworks supporting it, there is no reason not to use it if you are running PHP 5.3 or greater.

To install PHPUnit using Composer, once you've installed Composer itself, create a JSON file (commonly named composer.json) that tells Composer where you want it installed.

Here's an example that will install PHPUnit globally in the specified 'config' directory.

```
{
    "name": "phpunit",
    "description": "PHPUnit",
    "require": {
        "phpunit/phpunit": "3.7.*"
    },
    "config": {
        "bin-dir": "/usr/local/bin/"
    }
}
```

Composer will also try to pull in any required dependencies, but if for some reason they don't work, you can just add them to composer.json.

If you prefer to have PHPUnit as an actual dependency for your application, you can create a much simplified version of that JSON file

```
{
    "require-dev": {
        "phpunit/phpunit": "3.7.*"
    }
}
```

Then, if people choose to install or update your code via Composer and use the `--dev` flag, they will get the version of PHPUnit you have specified installed and available for use *inside* your project, in the vendor directory.

It's a tough call. On the one hand, you don't want to force people to install PHPUnit just to use your project. On the other hand, it does ensure that you can update PHPUnit independent of any other versions you already had installed.

Installing via PEAR

PEAR[3] used to be my preferred method of installation before they made it available via Composer. In this case, installing things can be as simple as:

```
path/to/pear config-set auto_discover 1
path/to/pear install pear.phpunit.de/PHPUnit
```

By default PEAR will try to pull in additional dependencies for PHPUnit, but you can manually install any additional missing components via PEAR as well.

[3] http://pear.php.net

Which One Should I Use

That depends on what additional dependencies you happen to be including. I definitely think you should choose one method and stick with it, however, as mixing Composer and PEAR might cause you to make mistakes and forget a package that you are likely to need. It's worth noting that Composer installs dependencies locally by default while PEAR installs globally by default.

In any case, consult the documentation for PHPUnit to see all the dependencies and add-ons that are available.

Minimum Viable Test Class

```
1  <?php
2  class GrumpyTest extends PHPUnit_Framework_TestCase
3  {
4      public function testMinimumViableTest()
5      {
6          $this->assertTrue(false, "true didn't end \
7  up being false!");
8      }
9  }
```

That is what I would call a Minimum Viable Test class. All test classes need to extend off of the base `PHPUnit_Framework_TestCase` class, although it is common for people to create their *own* base class that extends from this one, and then all their test cases extend from it.

All the built-in assertions that PHPUnit provides follow the same pattern:

- a type of assertion
- an expected value
- the value generated by the test
- an optional message to be displayed when the test fails

I can already tell that you are thinking "he's lying about the pattern because it says 'assertTrue.'" In a way, you are right. PHPUnit does provide some shortcuts to perform certain assertions. assertTrue() is one of them, along with its counterpart assertFalse(). These shortcuts do not change the fact that they all follow the same pattern.

For more details on all the assertions that are available to you, check the latest documentation[4].

Making Your Tests Tell You What's Failed

Verbose mode

You can add the option `--verbose` when running PHPUnit to get some more detailed information about the test run. Things like skipped or incomplete tests are important to know, especially if you are using multiple different configuration files and are purposely skipping some tests due to either how long they take to run or their flakiness due to use of outside sources of information.

[4] http://www.phpunit.de/manual/3.7/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.assertions

TestDox

A friend of mine related a story to me about an experience he had in a team he was working with:

> Today I was called in as the unit tests stopped somewhere in the beginning of the unit test run. I was really blown away to see the many E, F, I and S characters in the overview but I was more amazed about the fact that my team had no idea how to figure out which test caused the fatal error

Now, I mentioned that all these assertions take, as their final argument, an optional message to be displayed if the test fails. I cannot recommend highly enough that you make that message mandatory when you write your own tests. A descriptive message will go a long way towards helping you figure out exactly which test failed, hopefully telling you why.

There is also another option that doesn't involve using the optional message argument. You could use PHPUnit's TestDox[5] functionality. What it does is turn the name of your test methods into easily-read strings.

It will turn the test method name `testBankBalanceCannotGoIntoOverdraftU` into "Bank balance cannot go into overdraft unless allowed". But be careful: if you have tests that have the same name but you append an integer to the end, TestDox does not know that the two are different.

Here's a sample run using it.

[5] http://www.phpunit.de/manual/3.7/en/other-uses-for-tests.html

```
$ phpunit --testdox
PHPUnit 3.7.10 by Sebastian Bergmann.

Configuration read from /Users/chartjes/Sites/lies\
itoldmykids/tests/phpunit.xml

LieEntity
 [x] Description is not spammy
 [x] Description has swearing
 [x] Description has porn

LieMapper
 [x] Returns lie collection
 [x] Get one record
 [x] Get correctly handles not finding lie
 [x] Get valid lies
 [x] Create new lie
 [x] Delete known created entity
 [x] Delete correctly handles null lie entity id
 [x] Delete handles no deleted rows correctly
 [x] Delete handles missing entity correctly
 [x] Update known entity
 [x] Update correctly handles bad lie entity
 [x] Update correctly handles rejecting wrong lie \
update
```

TestDox simply gives you a human-readable list of the tests that have run. PHPUnit will still tell you what tests failed, but the TestDox version just might make it easier.

Configuring Run Time Options

If you run `phpunit --help` from the command line, you will see a ridiculous number of options that are available to you. Some of them are useful, others seem to be in there because the author was looking for compatibility with existing tools. Here are the ones that I have found most useful.

Code Coverage Options

Code coverage reports are a tool you can use to figure out how much of your code is being executed by your tests. This value is normally expressed as a percentage as in "I have 60% code coverage for my application… why is Chris glaring at me like that?"

You also can get what is known as the Change Risk Analysis and Predictions metric for your code. The CRAP score (I cannot think of a more appropriate acronym) is an indication of how complex it is. The higher the CRAP score, the more complicated your code is.

In order to generate a code coverage report, you need to have the PHP CodeCoverage component[6] and XDebug[7] installed.

There are two additional options you need to consider when generating code coverage:

Use `--coverage-clover optional/path/to/file` for generating Clover-formatted reports that can be read by Jenkins code coverage plugins.

[6] http://github.com/sebastianbergmann/php-code-coverage
[7] http://xdebug.org

Clover-formatted reports can be used to examine trends over time in terms of code coverage and lines of code added. These are extremely useful if you are trying to make sure developers are living up to their promises of writing tests with maximum code coverage.

Use `--coverage-html optional/path` to create a series of HTML files that you can view in your browser to see code coverage results.

Code coverage reports are also a great tool for code reviews – visible proof that you might be missing some tests for a few edge cases lurking deep inside your application.

Managing Global State

Many PHP applications make use of singletons that have static method calls, or rely on globals and super-globals (such as `$_SESSION` or `$_POST`, etc). While there are legitimate reasons from an architectural standpoint to use static methods, they are kryptonite when it comes to testing. Static classes, attributes, and variables are also considered part of the global state.

The problem? You often have no control over when and to what values these things can be set. PHPUnit offers you a few ways to handle this. First, by default, PHPUnit tries to run your tests in such a way that it isolates any changes made to global items, so you are somewhat covered there.

If you are using PHP 5.3 or greater, PHPUnit will give you the option to also backup and restore static attributes of user defined classes. Again, a good practice to allow for isolated tests.

My advice to you is avoid statics as much as possible. The backup-and-restore mechanism will increase memory usage and test execution time, and statics are all about trying to impose immutability in a language that likes everything to be dynamic.

However, if you do need to use them, here are some tips:

- `--no-globals-backup` will disable the default backup-and-restore $GLOBALS
- `--static-backup` will backup and restore static attributes by default
- `@backupGlobals` annotation can be used to temporarily disable the backup-and-restore functionality for globals
- `@backupStaticAttributes` does the same, but for static attributes

Test Environment Configuration

You'll find manually adding command-line switches when running your tests quickly becomes tedious. Fortunately, PHPUnit allows you to use a configuration file for specifying the default switches, among other settings. By default, PHPUnit will look for a file named either `phpunit.xml` or `phpunit.xml.dist` in the directory in which you run it, and use the values it contains to alter its own behavior.

You may need different configuration for different kinds of tests – e.g., unit tests vs. integration tests. PHPUnit allows you to indicate a specific configuration file using the `--configuration` switch, with an argument indicating the path of the configuration file.

To execute your tests with a specific configuration file:

```
/path/to/phpunit --configuration /path/to/your/php\
unit.xml
```

Command-Line Switches

Command line switches can be specified in the configuration file as
attributes of the root `phpunit` element. The following provides configuration for the `--backupGlobals` and `--processIsolation` switches,
respectively:

```
<phpunit
    backupGlobals="true"
    processIsolation="false">
<!-- other stuff goes here -->
</phpunit>
```

Appendix C[8] of the current PHPUnit documentation covers this in more detail. I highly recommend setting as many of the command-line switches as you can in the configuration file so you don't forget to do it yourself. Remember, computers are awesome at doing what you tell them to over and over again. Humans, not so much.

Process Isolation

If you've worked with PHP for any length of time, you become aware of the fact that there is global state, and that the state is preserved for the entire length of the request. In other

[8] http://www.phpunit.de/manual/current/en/appendixes.configuration.html

words, if you create an object at any point in the request, it will be available to any other code that resides in the same scope.

This can sometimes be a problem when running tests because you don't want state leaking from one test to another. Things could get unpredictable if you modify an object in one test when a subsequent test expects that object to be unmodified.

I usually see process isolation used in PHPUnit when running integration tests. Why? Integration tests usually consist of manipulating real objects, not test doubles, so you must pay close attention to their state.

To ensure process isolation for all your tests, it's as simple as passing `--process-isolation` as a CLI option, or setting `processIsolation="true"` in your XML configuration file. This means, by default, every single test will be run in it's own PHP process. This means your test suite will take a lot longer to run, so keep this in mind if you decide to do it.

If you need process isolation for all tests in a single file, you can enforce this by putting `@runInSeparateProcess` in the docblock for your test class.

If you only have some tests that need to be isolated, make sure to use the annotation in the docblock for the method. That way, it is only applied to that one method.

Another potential solution is to use separate `phpunit.xml` files that set process isolation as a run-time option and then white list only the directories containing the tests that need to be run in isolation. Conversely, make sure that your non-process-isolated configuration file doesn't include any tests that require isolation to work correctly.

As an example we have `phpunit-unit.xml`

```
<phpunit
    <!-- this is our config file for unit tests -->
    processIsolation="false">
<!-- other stuff goes here -->
</phpunit>
```

and we have `phpunit-integration.xml`

```
<phpunit
    <!-- this is our config file for integration t\
ests -->
    processIsolation="true">
<!-- other stuff goes here -->
</phpunit>
```

Then, when you are ready to run just your unit tests you run

```
path/to/phpunit --configuration path/to/phpunit-un\
it.xml
```

My experience has been that the multiple configuration file method is the best way to go if you are going to write unit and integration tests.

Organizing Your Tests

File System

The easiest way to organize your tests is via the file system. Create a directory for all your tests to run in, and PHPUnit will automatically recursively traverse the directories below your root test directory to find tests to run.

```
tests
|-- Foo
|   | -- Fizz
|   |    `-- InputsTest.php
|   | -- Buzz
|   |    `-- RecursionTest.php
|   `-- BazzTest.php
`-- FooTest.php
```

This is normally the way I organize my tests. Like with so many things related to programming, this is a personal preference.

Some developers prefer to bundle their tests alongside the code (think vendor directories when installing things via Composer). Some prefer all their tests to be in one flat directory.

Either way, it doesn't matter because PHPUnit will search all directories you ask it to looking for files with test cases in them.

Test Suites

In a previous section I talked about creating an XML test suite file. You can also specify exactly what tests you want run via XML. Let's create a file that mirrors the structure we used above.

```
<phpunit>
    <testsuites>
        <testsuite name="Foo">
            <file>Tests/Foo/Fizz/InputsTest.php</f\
ile>
            <file>Tests/Foo/Buzz/RecursionTest.php\
</file>
            <file>Tests/Foo/BazzTest.php</file>
            <file>Tests/FooTest.php</file>
        </testsuite>
    </testsuites>
</phpunit>
```

Test suites are a way of grouping tests together. This can be handy if you are making changes to some code and want to first just run the tests most likely to be impacted by the change.

To run the `Foo` test suite in the example file, you would do

```
/path/to/phpunit --testsuite Foo
```

You can also use the test suite functionality to "whitelist" your tests, ensuring that only the ones you want get executed. This is handy if you are working on a large team and have tests for code that is not quite finished yet and therefore don't need to see those test failures.

Want to group a test suite based on directory?

```
<phpunit>
    <testsuites>
        <testsuite name="Bar">
            <directory>alpha</directory>
            <directory>beta</directory>
            <directory>gamma</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

PHPUnit does not require you to define test suites in the configuration file.

Multiple Test Suites

As you start writing a large number of tests for your application, you'll observe some common problems:

- some of your tests might be flaky due to integration with 3rd party services you have not put wrappers around.
- shared testing database servers can often get overloaded, leading to transient test failures.

These things, while regrettable, are sometimes a reality for a resource-starved development team.

Through the use of test suites, you can create test plans that you can choose to execute on an as-needed basis. To use the above analogy, you might only want to run the flaky tests during the final set of tests before a production push.

So how do we do this? First, you'd define your test suites:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
    bootstrap="vendor/autoload.php"
    printerFile="vendor/whatthejeff/nyancat-phpuni\
t-resultprinter/src/NyanCat/PHPUnit/ResultPrinter.\
php"
    printerClass="NyanCat\PHPunit\ResultPrinter">
    <testsuites>
        <testsuite name="unit">
            <file>LieMapperTest.php</file>
            <file>UserMapperTest.php</file>
        </testsuite>
        <testsuite name="integration">
            <file>LieEntityTest.php</file>
        </testsuite>
    </testsuites>
</phpunit>
```

First, we run just the `unit` test suite

```
$ phpunit --testsuite unit                          \
                                                    \

PHPUnit 3.7.10 by Sebastian Bergmann.

Configuration read from /Users/chartjes/Sites/lies\
itoldmykids/tests/phpunit.xml

 23  -_-_-_-_-_-_-_-_-_-_-_-__,------,
 0   -_-_-_-_-_-_-_-_-_-_-_-__|  /\_/\
 0   -_-_-_-_-_-_-_-_-_-_-_-_~|_( ^ .^)
     -_-_-_-_-_-_-_-_-_-_-_-_ ""  ""
```

```
13
14
15  Time: 0 seconds, Memory: 4.25Mb
16
17  OK (23 tests, 74 assertions)
```

Then the `integration` test suite

```
1   $ phpunit --testsuite integration
2   PHPUnit 3.7.10 by Sebastian Bergmann.
3
4   Configuration read from /Users/chartjes/Sites/lies\
5   itoldmykids/tests/phpunit.xml
6
7    3   -_-__,------,
8    0   -_-__|  /\_/\
9    0   -_-_~|_( ^ .^)
10       -_-_ ""  ""
11
12
13  Time: 0 seconds, Memory: 3.75Mb
14
15  OK (3 tests, 6 assertions)
```

Confident that our code is fine, let's run them both

```
$ phpunit
PHPUnit 3.7.10 by Sebastian Bergmann.

Configuration read from /Users/chartjes/Sites/lies\
itoldmykids/tests/phpunit.xml

 26  -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_,------,
 0   -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-|   /\_/\
 0   -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-^|__( ^ .^)
     -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-  ""  ""


Time: 0 seconds, Memory: 4.50Mb

OK (26 tests, 80 assertions)
```

Alternate Test Runners

By default, PHPUnit's test runner is pretty bland. It doesn't even do red for failures or green for passes! Being up on Internet memes, I like to use a Nyan Cat[9] test runner to display my results.

To enable it, I installed the test runner[10] using Composer and then altered my PHPUnit configuration file to use it.

[9] http://www.nyan.cat/

[10] https://github.com/whatthejeff/nyancat-phpunit-resultprinter/

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
    bootstrap="vendor/autoload.php"
    printerFile="vendor/whatthejeff/nyancat-phpuni\
t-resultprinter/src/NyanCat/PHPUnit/ResultPrinter.\
php"
    printerClass="NyanCat\PHPunit\ResultPrinter">
    <testsuites>
        <testsuite name="Lies I Told My Kids">
            <file>LieEntityTest.php</file>
            <file>LieMapperTest.php</file>
            <file>UserMapperTest.php</file>
            <exclude>./vendor</exclude>
            <exclude>./report</exclude>
        </testsuite>
    </testsuites>
</phpunit>
```

Behold, the Nyan Cat!

```
$ phpunit
PHPUnit 3.7.10 by Sebastian Bergmann.

Configuration read from /Users/chartjes/Sites/lies\
itoldmykids/tests/phpunit.xml

 26  -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_,------,
 0   -_-_-_-_-_-_-_-_-_-_-_-_-_-_-_|   /\_/\
 0   -_-_-_-_-_-_-_-_-_-_-_-_-_-_-^|__( ^ .^)
     -_-_-_-_-_-_-_-_-_-_-_-_-_-_-  ""  ""


```

```
13 Time: 1 second, Memory: 4.50Mb
14
15 OK (26 tests, 79 assertions)
```

Test Doubles

Why We Need Them

When you are writing unit tests, you are writing tests to create scenarios where you control the inputs and want to verify that your code is returning expected results.

Code you write often depends on other code: things like database handles or values from a globally-available registry. In order to write tests for code using these dependencies, you need to be able to set those dependencies to specific states in order to predict the expected result.

One way to do this is through the use of dependency injection[11]. That Wikipedia article is long and technical, but the lesson to be learned is that you should be creating small modules of code that accept their dependencies one of three ways:

- passing them in via an object's constructor method and assigning them to class attributes
- by assigning dependencies to class attributes directly or by setters
- by using a dependency injection container that is globally available

[11] http://en.wikipedia.org/wiki/Dependency_injection

Whatever method you choose (I prefer constructor injection), you can create objects in a known state through the use of test doubles created using PHPUnit's built-in mocking functionality.

What Are They

In the pure testing world, there are five types of test doubles:

- dummy objects
- test stubs
- test spies
- test mocks
- test fakes

PHPUnit is not a purist in what it does. Dummy objects, stubs and mocks are available out of the box, and you can kind of, sort of, create spies using some of the methods provided by the mocking API. No test fakes either.

Dummy Objects

```php
<?php
class Baz
{
    public $foo;
    public $bar;

    public function __construct(Foo $foo, Bar $bar)
    {
        $this->foo = $foo;
        $this->bar = $bar;
    }

    public function processFoo()
    {
        return $this->foo->process();
    }

    public function mergeBar()
    {
        if ($this->bar->getStatus() == 'merge-read\
y') {
            $this->bar->merge();
            return true;
        }

        return false;
    }
}
```

Sometimes we just need something that can stand in for a dependency and we are not worrying about faking any functionality. For this we use a dummy object.

Looking at the code above, in order to test anything we need to pass in a Foo and Bar object.

```
<?php
public function testThatBarMergesCorrectly()
{
    $foo = $this->getMockBuilder('Foo')->getMock();

    $bar = $this->getMockBuilder('Bar')
        ->setMethods(array('getStatus', 'merge'))
        ->getMock();
    $bar->expects($this->once())
        ->method('getStatus')
        ->will($this->returnValue('merge-ready'));

    // Create our Baz object and then test our fun\
ctionality
    $baz = new Baz($foo, $bar);
    $expectedResult = true;
    $testResult = $baz->mergeBar();

    $this->assertEquals(
        $expectedResult,
        $testResult,
        'Baz::mergeBar did not correctly merge our\
 Bar object'
    );
}
```

Our dummy object in this test is the `Foo` object we created. This test doesn't care if `Foo` does anything. Remember, the goal when writing tests is to also minimize the amount of

testing code you need to write. Don't create test doubles for things if they aren't needed for the test!

Test Stubs

```
<?php
public function testMergeOfBarDidNotHappen()
{
    $foo = $this->getMockBuilder('Foo')->getMock();
    $bar = $this->getMockBuilder('Bar')->getMock();
    $bar->expects($this->any())
        ->method('getStatus')
        ->will($this->returnValue('pending'));

    $baz = new Baz($foo, $bar);
    $testResult = $baz->mergeBar();

    $this->assertFalse(
        $testResult,
        'Bar with pending status should not be mer\
ged'
    );
}
```

A test stub is a mock object that you create (remember, a dummy object is really a mock object without any functionality) and then alter it so that when specific methods are called, we get a specific response back.

In the test above, we are creating a test stub of `Bar` and controlling what a call to `getStatus` will do.

A word of warning: PHPUnit cannot mock protected or private class methods. To do that you need to use PHP's

Reflection API[12] to create a copy of the object you wish to test and set those methods to be publicly visible.

I realize that from a code architecture point of view protected and private methods have their place. As a tester, they are a pain.

To end the stub method, we use `will` to tell the stubbed method what we want it to return.

Understanding how to stub methods in the dependencies of the code you are trying to test is the number one skill that good testers learn.

You will learn to use test stubs as a "code smell", since it will reveal that you might have too many dependencies in your code, or that you have an object that is trying to do too much. Inception-level test stubs inside test stubs inside test stubs is an indication that you need to do some rethinking of your architecture.

Expectations during execution

Here is an example of a test that sets expectations for what a particular method should return when called multiple times.

[12] http://php.net/manual/en/book.reflection.php

```
<?php
public function testShowingUsingAt()
{
    $foo = $this->getMockBuilder('Foo')->getMock();
    $foo->expects($this->at(0))
        ->method('bar')
        ->will($this->returnValue(0));
    $foo->expects($this->at(1))
        ->method('bar')
        ->will($this->returnValue(1));
    $foo->expects($this->at(2))
        ->method('bar')
        ->will($this->returnValue(2));
    $this->assertEquals(0, $foo->bar());
    $this->assertEquals(1, $foo->bar());
    $this->assertEquals(2, $foo->bar());
}
```

There are a number of values that we can use for `expects()`:

- `$this->at()` can be used to set expected return values based on how many times the method is run. It accepts an integer as a parameter and starts at 0
- `$this->any()` won't care how many times you run it, and is the choice of lazy testers everywhere
- `$this->never()` expects the method to never run
- `$this->once()` expects the method to be called only once during the test. If you are using `$this->with` that we talk about in the next section, PHPUnit will check that the method is called once with those specific parameters being passed in

- `$this->exactly()` expects the method to be called a specific number of items. It accepts an integer as the parameter
- `$this->atLeastOnce()` is an interesting one, a good alternative to any()

When you are creating expectations for multiple calls, be aware that whatever response you are setting through the use of `this->with` is only applicable to that specific expectation.

Returning Specific Values Based On Input

```
<?php
public testChangingReturnValuesBasedOnInput()
{
    $foo = $this->getMockBuilder('Foo')->getMock();
    $foo->expects($this->once())
        ->method('bar')
        ->with('1')
        ->will($this->returnValue('I'));
    $foo->expects($this->once())
        ->method('bar')
        ->with('4')
        ->will($this->returnValue('IV'));
    $foo->expects($this->once())
        ->method('bar')
        ->with('10')
        ->will($this->returnValue('X'));
    $expectedResults = array('I', 'IV', 'X');
    $testResults = array();
    $testResults[] = $foo->bar(1);
```

```
20      $testResults[] = $foo->bar(4);
21      $testResults[] = $foo->bar(10);
22
23      $this->assertEquals($expectedResults, $testRes\
24  ults);
25  }
```

Often, the methods you invoke on dependencies accept arguments, and will return different values depending on what was passed in. With PHPUnit, you can tell a mock object what to expect for arguments, and bind a specific return value for that input. To do this, you use the `with()` method to detail arguments, and the `returnValue()` method, inside the `will()` method, to detail return values.

The test above shows you how to go about creating the expectation of a specific result based on a specific set of input parameters.

Mocking method calls with multiple parameters

If you need to mock a method that accepts multiple parameters, you can specify that inside the `with()` method; specify the arguments in the same order the method accepts them:

```
<?php
// Method is fooSum($startRow, $endRow)
$foo = $this->getMockBuilder('MathStuff')->getMock\
();
$startRow = 1;
$endRow = 10;
$foo->expects($this->once())
    ->method('fooSum')
    ->with($startRow, $endRow)
    ->will($this->returnValue(55));
```

Testing protected and private methods and attributes

I will be up front: I am not an advocate of writing tests for protected and private class methods. In most cases, when you write tests for public methods, you will end up testing private and protected methods that it calls.

Of course, if these private and protected methods have side effects, you will probably need to test them just to make sure they are tested properly.

In PHP, the only way to do this easily is by using the Reflection API that is available in PHP 5.

Here is a very contrived example:

```
<?php
class Foo
{
    protected $message;

    protected function bar($environment)
    {
        $this->message = "PROTECTED BAR";

        if ($environment == 'dev') {
            $this->message = 'CANDY BAR';
        }

    }
}
```

To create a test double that we can use, we need to follow these steps:

- use the Reflection API to create a reflected copy of the object
- call `setAccessible(true)` on the method in the reflected object you want to test
- call `invoke()` on the reflected object, passing in the name of the method to test and any parameters you wish to pass in

If you need to perform an assertion against the contents of a protected or private attribute, you can then use `assertAttribute` just like you would do any other assertion. Chapter 4[13] of

[13] http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.assertions

the PHPUnit documentation covers all the different types of assertions you can make.

```
<?php
class FooTest extends PHPUnit_Framework_TestCase
{
    public function testProtectedBar()
    {
        $expectedMessage = 'PROTECTED BAR';
        $reflectedFoo = new ReflectionMethod('Foo'\
, 'bar');
        $reflectedFoo->setAccessible(true);
        $reflectedFoo->invoke(new Foo(), 'producti\
on');

        $this->assertAttributeEquals(
            $expectedMessage,
            'message',
            $reflectedFoo,
            'Did not get expected message'
        );
    }
}
```

Test Spies

The main difference between a test spy and a test stub is that you're not concerned with testing return values, only that a method you are testing has been called a certain number of times.

Given the code below:

```php
<?php
class Alpha
{
    protected $beta;

    public function __construct($beta)
    {
        $this->beta = $beta;
    }

    public function cromulate($deltas)
    {
        foreach ($deltas as $delta) {
            $this->beta->process($delta);
        }
    }

    // ...
}
```

If you wanted to create test spies to make sure a method is executed 3 times, you could do something like this:

```php
<?php

// In your test class...
public function betaProcessCalledExpectedTime()
{
    $beta = $this->getMockBuilder('Beta')->getMock\
();
    $beta->expects(3)->method('process');
    $deltas = array(-18.023, -14.123, 3.141);
```

```
10
11      $alpha = new Alpha($beta);
12      $alpha->cromulate($deltas);
13  }
```

In this code, we want to make sure that, given a known number of 'deltas', `process()` gets called the correct number of times. In the above test example, the test case would fail if `process()` is not run 3 times.

As you saw in earlier tests in this chapter, `expects()` can accept a wide variety of options, but you can pass it an integer that matches the number of times you are expecting the method to be called in your test.

More Object Testing Tricks

Testing Traits

Because Traits can be defined once, but used many times, you will not want to necessarily test the functionality defined in traits in every object in which they are consumed. At the same time, you do want to test the traits themselves.

PHPUnit 3.6 and newer offers functionality for mocking traits via the `getObjectForTrait()` method; this will return an object composing the trait, so that you can unit test only the trait itself.

Here's a sample trait:

```
<?php
trait Registry
{
    protected $values = array();

    public function set($key, $value)
    {
        $this->values[$key] = $value;
    }

    public function get($key)
    {
        if (!isset($this->values[$key])) {
            return false;
        }

        return $this->values[$key];
    }
}
```

Okay, now we test it

```
<?php
class RegistryTest extends PHPUnit_Framework_TestC\
ase
{
    public function testReturnsFalseForUnknownKey()
    {
       $registry = $this->getObjectForTrait('Regis\
try');
        $response = $registry->get('foo');
        $this->assertFalse(
```

```
11              $response,
12              "Did not get expected false result for\
13  unknown key"
14          );
15      }
16  }
```

Data Providers

Why You Should Use Data Providers

One of your main goals should always be to write the bare minimum amount of code in order to solve a particular problem you are facing. This is no different when it comes to tests, which are really nothing more than code.

One of the earliest lessons I learned when I first started writing what I felt were comprehensive test suites was to always be on the lookout for duplication in your tests. Here's an example of a situation where this can happen.

Most programmers are familiar with the FizzBuzz[14] problem, if only because it is commonly presented as a problem to be solved as part of an interview. In my opinion it is a good problem to present because it touches on a lot of really elementary basics of programming.

When you write tests for FizzBuzz, what you want to do is pass it a set of values and verify that they are FizzBuzzed correctly. This could result in you having multiple tests that are the same except for the values you are testing with. Data providers give you a way to simplify that process.

A data provider is a way to create multiple sets of testing data that can be passed in as parameters to your test method. You

[14] http://en.wikipedia.org/wiki/FizzBuzz

create a method that is available to the class your tests are in that returns an array of values that match the parameters that you are passing into your test.

I know it sounds more complicated than it really is. Let's look at an example.

Look At All Those Tests

If you didn't know about data providers, what might your FizzBuzz tests look like?

```
<?php
class FizzBuzzTest extends PHPUnit_Framework_Testc\
ase
{
    public function setup()
    {
        $this->fb = new FizzBuzz();
    }

    public function testGetFizz()
    {
        $expected = 'Fizz';
        $input = 3;
        $response = $this->fb->check($input);
        $this->assertEquals($expected, $response);
    }

    public function testGetBuzz()
    {
        $expected = 'Buzz';
```

```
21          $input = 5;
22          $response = $this->fb->check($input);
23          $this->assertEquals($expected, $response);
24      }
25
26      public function testGetFizzBuzz()
27      {
28          $expected = 'FizzBuzz';
29          $input = 15;
30          $response = $this->fb->check($input);
31          $this->assertEquals($expected, $response);
32      }
33
34      function testPassThru()
35      {
36          $expected = '1';
37          $input = 1;
38          $response = $this->fb->check($input);
39          $this->assertEquals($expected, $response);
40      }
41  }
```

I'm sure you can see the pattern:

- multiple input values
- tests that are extremely similar in setup and execution
- same assertion being used over and over

Creating Data Providers

A data provider is another method inside your test class that returns an array of results, with each result set being an

array itself. Through some magic internal work, PHPUnit converts the returned result set into parameters which your test method signature needs to accept.

```
<?php
public function fizzBuzzProvider()
{
    return array(
        array(1, '1'),
        array(3, 'Fizz'),
        array(5, 'Buzz'),
        array(15, 'FizzBuzz')
    );
}
```

The function name for the provider doesn't matter, but use some common sense when naming them as you might be stumped when a test fails and tells you about a data provider called 'ex1ch2', or something else equally meaningless.

To use the data provider, we have to add an annotation to the docblock preceding our test so that PHPUnit knows to use it. Give it the name of the data provider method.

```php
<?php
/**
 * Test for our FizzBuzz object
 *
 * @dataProvider fizzBuzzProvider
 */
public function testFizzBuzz($input, $expected)
{
    $response = $this->fb->check($input);
    $this->assertEquals($expected, $response);
}
```

Now we have just one test (less code to maintain) and can add scenarios to our heart's content via the data provider (even better). We have also learned the skill of applying some critical analysis to the testing code we are writing to make sure we are only writing the tests that we actually need.

When using a data provider, PHPUnit will run the test method each time for every set of data being passed in by the provider. If the test fails it will indicate which index in the associative array was being used for that test run.

More Complex Examples

Don't feel like you can only have really simple data providers. All you need to do is return an array of arrays, with each result set matching the parameters that your testing method is expecting. Here's a more complex example:

```
<?php
public function complexProvider()
{
    // Read in some data from a CSV file
    $fp = fopen("./fixtures/data.csv");
    $response = array();

    while ($data = fgetcsv($fp, 1000, ",")) {
        $response[] = array($data[0], $data[1], $d\
ata[2]);
    }

    fclose($fp);

    return $response;
}
```

So don't think you need to limit yourself in what your data providers are allowed to do. The goal is to create useful data sets for testing purposes.

Data Provider Tricks

Since data providers return associative arrays, you can assign them a more descriptive key to help with debugging. For example, we could refactor the data provider for our FizzBuzz test:

```
<?php
return array(
    'one'      => array(1, '1'),
    'fizz'     => array(3, 'Fizz'),
    'buzz'     => array(5, 'Buzz'),
    'fizzbuzz' => array(15, 'FizzBuzz')
);
```

Also, data providers don't have to be methods inside the same class. You can use methods in other classes, you just have to make sure to define them as `public`. You can use namespaces as well. Here are two examples:

- `@dataProvider Foo::dataProvider`
- `@dataProvider Grumpy\Helpers\Foo::dataProvider`

This allows you to create helper classes that are just data providers and cut down on the amount of duplicated code you have in your tests themselves.

Creating Test Data

If you are testing functionality that needs data to manipulate, you need to learn how to create and provide realistic data for your tests.

Note the key word here is "realistic." Even the best tests are of no use to you if they consume data that is wildly different from the data your code uses in production.

Data Source Snapshots

If you are using PHPUnit to do integration tests, or haven't really written unit tests that use test doubles to simulate speaking to a data source, then investing time in scripts that can take snapshots of your data will pay off.

Shell scripts

One of the easiest ways I know to get large amounts of data out of a data source easily is to use whatever CLI utilities are provided with it. The classic example is using *mysql_dump* to grab tables and then the MySQL CLI tool to import the data into the database associated with your tests.

Any tool that can automate the process of getting the raw data from one location to another will be of use to you.

Serialize And Store

I have a personal preference for architectures where I speak to data sources and create data objects from the results. One benefit of this is that, during testing, I can create collections of objects by writing some PHP code and serializing the results to the file system for later retrieval.

Let's say that we want to create a collection of Widgets. They are simple data objects.

```
<?php
class Widget
{
    public $sku;
    public $description;
    public $color;
```

Because `Widget` has no hidden dependencies or dependencies on other resources it is a prime candidate to be serialized.

Next, we can write a script to generate our serialized collection of objects.

```
<?php
// $dbh is a PDO DB object
$sql = "SELECT * FROM widgets WHERE type = 'standa\
rd'";
$stmnt = $dbh->prepare($sql);
$stmnt->execute();
$results = $stmnt->fetchAll(PDO::FETCH_ASSOC);
$collection = array();

```

```
foreach ($results as $result) {
    $widget = new Widget();
    $widget->sku = $result['sku'];
    $widget->description = $result['description'];
    $widget->color = $result['color'];
    $widget->price = $result['price'];
    $collection[] = $widget;
}

$data = serialize($collection);
file_put_contents(
    './tests/fixtures/widget-collection.txt',
    $data
);
```

When you're ready to use it in your test:

```
<?php
$data = file_get_contents('./test/fixtures/widget-\
collection.txt');
$collection = unserialize($data);

$widgetGrouper = new WidgetGrouper($collection);
$expectedCount = 7;
$testCount = $widgetGrouper->getCountByColor('blac\
k');

$this->assertTrue(
    count($collection) > 8,
    "Did not have at least 8 widgets in our collec\
tion"
);
```

```
$this->assertEquals(
    $expectedCount,
    $testCount,
    "Did not get expected count of black widgets"
);
```

Fake It When You Need To

Remember I pointed out earlier in the chapter that it's good to have realistic testing data? When you write tests, it's very tempting to take short cuts. For instance, I have been known to overuse such famous people as Testy McTesterton and Art Vandelay as test subjects.

Faker[15] is a great tool for randomly generating data like names, addresses, and phone numbers.

```
<?php
$faker = Faker\Factory::create();
$foo = new Foo();
$foo->name = $faker->name;
$foo->address1 = $faker->streetAddress;
$foo->address2 = null;
$foo->city = $faker->city;
$foo->state = $faker->state;
$foo->zip = $faker->postcode;

// $dbh is our PDO database handler
$fooMapper = new FooMapper($dbh);
$fooMapper->create($foo);
```

[15] https://github.com/fzaninotto/Faker

Faker has a ridiculous number of options available to you, too many to list here. A few highlights:

- localization abilities
- different types of emails
- date and time values
- user agents
- ORM integration (Propel, Doctrine)
- create your own data providers

Testing API's

```php
<?php
namespace Grumpy;

class GimmebarApi
{
    protected $apiUrl;

    public function __construct($apiUrl)
    {
        $this->apiUrl = $apiUrl;
    }

    public function getPublicAssetCountByUser($use\
rname)
    {
        $response = $this->grabPublicAssetsByUser(\
$username);

        if (isset($response['total_records'])) {
            return $response['total_records'];
        }

        return false;
    }

    public function grabPublicAssetsByUser($userna\
me)
```

```
28      {
29          $response = file_get_contents(
30              $this->apiUrl . "/public/assets/{$user\
31  name}"
32          );
33
34          return json_decode($response, true);
35      }
36  }
```

If there is one question I get over and over again from people seeking testing advice, it's "how can I test API calls?" The only question that I get asked more is "why are you so grumpy all the time?"

Testing API calls is really no different than testing any other kind of code: you have an expected output, you execute some code that calls the API, you verify your test returns the values that you are expecting.

Testing Talking To The API Itself

The most common test that people are looking to do is one where you get data from an API and then you want to transform that data somehow. Given our example code, it might look something like this:

```
<?php
public function testCountPublicAssetsByKnownUser()
{
    $apiUrl = 'https://gimmebar.com/api/v1';
    $gimmebar = new \Grumpy\GimmebarApi($apiUrl);
    $expectedResultCount = 86;
    $publicAssetCount = $gimmebar->getPublicAssetC\
ountByUser('grumpycanuck');
    $this->assertEquals(
        $expectedResultCount,
        $publicAssetCount,
        'Did not get expected public asset count'
    );
}
```

This test is brittle because it relies on the API being available at the exact time we run the test. What happens if you can't actually reach this API from your testing environment? This becomes important if you are rate-limited in your access.

Speaking directly to the API also reduces your ability to be 100% certain that the API will return what you expect. Make sure to do periodic checks that the API's you are using are still returning values you expect, or else you will end up with tests that do not reflect reality.

Wrapping Your API Calls

I know this book is supposed to be about using PHPUnit, not about what your code is supposed to look like. Nonetheless I still think it's important to understand that the key to really being able to test APIs is wrapping code around how you access it.

By this I mean that if you have a library that someone (maybe even you) wrote to speak to an API, you really should be creating a wrapper around that access. Why? For testing purposes, of course!

This does mean that we will have to refactor the code so that our API class just returns a raw JSON response, and then we create a wrapper object that manipulates the API responses.

Yes, this sucks. But testable code is the key.

First, let's refactor our API object:

```
<?php
namespace Grumpy;

class GimmebarApi
{
    protected $apiUrl;

    public function __construct($apiUrl)
    {
        $this->apiUrl = $apiUrl;
    }

    public function grabPublicAssetsByUser($userna\
me)
    {
        $response = file_get_contents(
            $this->apiUrl . "/public/assets/{$user\
name}"
        );

        return json_decode($response, true);
```

```
22      }
23  }
```

Next, we create a wrapper object that takes the response from the API object and then applies some transformations to it. This way, the wrapper doesn't actually need to know that it is not dealing with the real thing. It just knows it's getting something that it knows how to use.

To isolate code for testing purposes, we should then proceed to create a mock object representing the real API, and pass that into our wrapper object.

```
1   <?php
2   namespace Grumpy;
3
4   class GimmebarWrapper
5   {
6       protected $_api;
7
8       public function __construct($api)
9       {
10          $this->_api = $api;
11      }
12
13      public function grabPublicAssetsByUser($userna\
14  me)
15      {
16          return $this->_api->grabPublicAssetsByUser\
17  ($username);
18      }
19
20      public function grabPublicAssetCountByUser($us\
```

```
ername)
    {
        $response = $this->_api->grabPublicAssetsB\
yUser($username);

        if (isset($response['total_response'])) {
            return $response['total_response'];
        }

        return false;
    }
}
```

Okay, so now that we have a wrapper that accepts our newly refactored API object, let's write a test to verify that stuff is working the way that we expect.

```
<?php
public function testCorrectlyFormatResponseFromApi\
()
{
    // Sample JSON response from Gimmebar API
    $apiResponse = <<<EOT
    {
       "more_records": true,
       "records":[{
          "id":"506e217c29ca15dc58000025",
          "asset_type":"page",
          "content":{
             "full":"http:\/\/gimmebar-assets.s3.a\
mazonaws.com\/506e217a50d77.html",
             "original":"http:\/\/merrickchristens\
```

```
en.com\/articles\/javascript-dependency-injection.\
html",
             "fullscreen":"http:\/\/s3.amazonaws.c\
om\/gimme-grabs-new\/18AF8B43-E8E5-40F9-BA4C-92FC4\
ADA9DA1",
             "thumb":"http:\/\/s3.amazonaws.com\/g\
imme-grabs-new\/CACED21F-2D18-453F-AEBF-E4B79802E8\
8C"
          },
          "date":1349394812,
          "media_hash":"c1cdc3e3fb3da302162ecb990a\
0dfa9216217",
          "private":false,
          "source":"http:\/\/merrickchristensen.co\
m\/articles\/javascript-dependency-injection.html"\
,
          "description":"",
          "tags":[
          ],
          "size":16217,
          "mime_type":"text\/html",
          "username":"grumpycanuck",
          "user_id":"b7703f2a12d4c7e1cc2f6999e593e\
3d0",
          "title":"Merrick Christensen - JavaScrip\
t Dependency Injection",
          "short_url":"http:\/\/gim.ie\/3PxM"
         }],
       "total_records": 1,
       "limit": 10,
       "skip": 0
    }
```

```
48 EOT;
49
50     $api = $this->getMockBuilder('\Grumpy\Gimmebar\
51 Api')
52         ->setMethods(array('grabPublicAssetsByUser\
53 '))
54         ->getMock();
55     $api->expects($this->once())
56         ->method('grabPublicAssetsByUser')
57         ->will($this->returnValue(json_decode($api\
58 Response,true)));
59
60     $apiWrapper = new \Grumpy\GimmebarWrapper($api\
61 );
62     $testResponse = $apiWrapper->grabPublicAssetsB\
63 yUser('chartjes');
64     $this->assertTrue(is_array($testResponse));
65 }
```

In this test case we are making sure that the Gimmebar wrapper is correctly handling a typical response we'd get from Gimmebar itself. Here's another example of a test using a mock object:

```
<?php
public function testCorrectlyCountPublicAssets()
{
    // Sample JSON response from Gimmebar API
    $apiResponse = "{'total_records': 10}";
    $api = $this->getMockBuilder('\Grumpy\Gimmebar\
Api')
        ->setMethods(array('grabPublicAssetsByUser\
'))
        ->getMock();
    $api->expects($this->once())
        ->method('grabPublicAssetsByUser')
        ->will($this->returnValue($apiResponse);

    $apiWrapper = new \Grumpy\GimmebarWrapper($api\
);
    $expectedCount = 10;
    $testCount = $apiWrapper->getPublicAssetCountB\
yUser('test');

    $this->assertEquals(
        $expectedCount,
        $testCount,
        "Did not correctly count the number of pub\
lic assets"
    );
```

Just like any other test, we're still following the same logic: we create a scenario, mock out resources that are required for that scenario, and then test our code to make sure that, based on a known set of inputs, we are getting an expected output.

Pay attention to the fact that in order to test this particular

bit of functionality, we don't even need a full response. Just a fake response containing only the data required is all it takes.

Testing Databases

```php
<?php
namespace Grumpy;

class Roster
{
    protected $db;

    public function __construct($db)
    {
        $this->db = $db;
    }

    public function getByTeamNickname($nickname)
    {
        $sql = "
            SELECT tig_name
            FROM rosters
            WHERE ibl_team = ?";
        $sth = $this->db->prepare($sql);
        $sth->execute(array($nickname));
        $rows = $sth->fetchAll(PDO::FETCH_ASSOC);

        if (!$rows) {
            return array();
        }

        $rosterContents = array();
```

```
28
29          foreach ($rows as $row) {
30              array_push($rosterContents, $row['tig_\
31  name']);
32          }
33
34          return $rosterContents;
35      }
36  }
```

Functional Tests vs. Unit Tests

Most web applications are thin wrappers around a database, despite our attempts to make them sound a lot more complicated than that. If we have code that speaks to a database, we need to be testing it.

Before we go any further, I want to make a distinction about types of tests. If you want to be strict about how you are defining your tests, then if you are writing unit tests, you should never be speaking to the database.

Why? Unit test suites are meant to be testing code, not the ability of a database server to return results. They also need to run quickly. If your test suite takes a long time to run, nobody is going to bother running it. Who wants to wait 30 minutes for your entire test suite to run? I sure don't.

If you are testing code that does complex database queries, guess what? Your test will be waiting every single time you run it for that query to finish. Again, all those little delays in the execution time for your test suite add up to people becoming more and more reluctant to run the entire test suite. This leads to bugs crossing "units" being discovered

only when someone runs the whole test suite. That's not good enough. We can do better.

I advocate writing as few tests as possible that speak directly to the database. If you have some business logic for your application that exists only in an SQL query, then you probably will have to write a few tests that speak to the database directly.

After all, I am only interested in testing to see if I can connect to my database properly. That sort of thing should be written into your application way before any business logic code gets run. Like in the bootstrap or the front controller of your framework-based code base.

Sandboxes

If you are going to write tests that connect to a database, then make sure you create a sandbox that the database will live in. When I say sandbox, I am referring to creating an environment where you can delete and recreate the database easily; automating these tasks is even better.

Make sure that your application supports the ability to decide what database it will talk to. Set it in the bootstrap, or in your globally-available configuration object, or in the constructor of the base class every other object in your application extends itself from. I don't care, just make sure that you can tell your application what database to talk to.

Not to beat a dead horse, but code that you can inject your dependencies into makes testing database-driven code a lot easier.

A sample test that talks directly to the database:

```
<?php
class RosterDBTest extends PHPUnit_Framework_TestC\
ase
{
    protected $db;

    public function setUp()
    {
        $dsn = "pgsql:host=127.0.0.1;dbname=ibl_st\
ats_test;user=stats;password=********";
        $this->db = new PDO($dsn);
    }

    public function testReturnsRosterSummaryForKno\
wnRoster()
    {
        $roster = new \Grumpy\Roster($this->db);
        $expectedRoster = array('AAA Foo', 'BBB Ba\
r', 'ZZZ Zazz');
        $testRoster = $roster->getByTeamNickname('\
TEST');
        $this->assertEquals(
            $expectedRoster,
            $testRoster,
            "Did not get expected roster when pass\
ing in known team nickname"
        );
    }
}
```

The downside to writing tests that speak directly to a database is that you end up needing to constantly maintain a testing database. If you have a testing environment where multiple

developers share the same database, you run a real risk of over-writing test data or even ending up with datasets that bear no resemblance to data that actually exists in production.

Like with any kind of testing, you are looking to compare expected results to the output of your code, given a known set of inputs. The only way to really achieve this is to either have your testing process dump the existing database and recreate it from scratch, or use database fixtures.

In the PHPUnit world, I feel there is only one database-fixture-handling tool worth considering: DBUnit.

DBUnit

As I've said before, I am not a big fan of using database fixtures, instead preferring to write my code in such a way that I instead create objects to represent the data: Easier to mock, easier to test. But you are not me. If you want to use a database in your tests, I recommend the use of DBUnit[16].

Check the web site for installation details. As of this writing it can be installed via PEAR or Composer.

There are times when you do need to talk to a database as part of a test, usually to verify that if you are saving some information to the database that it is still there. Let's look at a way we can create a test that involves speaking to the database.

Setting Things Up For DBUnit

Instead of extending from the usual PHPUnit test case object, we need to use a different one, and implement two required

[16] https://github.com/sebastianbergmann/dbunit

methods.

Using our sample app, here's one way to do it.

```
<?php
class RosterDBTest extends PHPUnit_Extensions_Data\
base_Testcase
{
    public function getConnection()
    {
        $dsn = "pgsql:host=127.0.0.1;dbname=ibl_st\
ats;user=stats;password=*********";
        $pdo = new PDO($dsn);

        return $this->createDefaultDBConnection($p\
do, $dsn);
    }

    public function getDataSet()
    {
        // Load your dataset here
    }

    // Existing tests go below
}
```

These two methods we've implemented make sure that any calls to a database being accessed via PDO will be intercepted by DBUnit.

When creating your connection in the `getConnection()` method, make sure to use the same database credentials that your application is expecting. Otherwise DBUnit won't intercept calls to the database.

In testing terms, the file that you put in the data you wish to be loaded is called a fixture. When organizing my testing code I like to create a directory called `fixtures` and put all of them in there.

Using XML Datasets

DBUnit supports several types of XML data fixtures. For my tests that do use them, I like to use "flat XML datasets".

Here's an example XML file that I put into `fixtures/roster-seed.xml`

```
<?xml version="1.0" ?>
<dataset>
        <rosters id="1" tig_name="FOO Bat" ibl_tea\
m="MAD" comments="Test record" status="0" item_typ\
e="2" />
        <rosters id="2" tig_name="TOR Bautista" ib\
l_team="MAD" comments="Joey bats!" status="1" item\
_type="1" />
        <rosters id="3" tig_name="MAD#1" ibl_team=\
"MAD" status="0" comments="Draft Pick" item_type="\
0" />
        <rosters id="4" tig_name="TOR Hartjes" ibl\
_team="MAD" comments="Test writer" status="1" item\
_type="1" />
</dataset>
```

Then you load it like this:

```
<?php
public function getDataSet()
{
    return $this->createFlatXMLDataset(
        dirname(__FILE__) . '/fixtures/roster-seed\
.xml');
}
```

If you prefer to be more of a purist, you could create a structured XML dataset. For our sample dataset, it would look like this:

```
<?xml version="1.0" ?>
<dataset>
    <table name="rosters">
        <column>id</column>
        <column>tig_name</column>
        <column>ibl_team</column>
        <column>comments</column>
        <column>status</column>
        <column>item_type</column>
        <row>
            <value>1</value>
            <value>FOO Bat</value>
            <value>MAD</value>
            <value>Test record</value>
            <value>0</value>
            <value>2</value>
        </row>
        <row>
            <value>2</value>
            <value>TOR Bautista</value>
```

```
            <value>MAD</value>
            <value>Joey bats!</value>
            <value>1</value>
            <value>1</value>
        </row>
        <row>
            <value>3</value>
            <value>MAD#1</value>
            <value>MAD</value>
            <value>Draft pick</value>
            <value>0</value>
            <value>0</value>
        </row>
        <row>
            <value>4</value>
            <value>TOR Hartjes</value>
            <value>MAD</value>
            <value>Test writer</value>
            <value>1</value>
            <value>1</value>
        </row>
    </table>
</dataset>
```

Loading that dataset is very similar:

```
<?php
public function getDataSet()
{
    return $this->createXMLDataset(
        dirname(__FILE__) . '/fixtures/roster-seed\
.xml');
}
```

One of the drawbacks to using an XML dataset is that if you have null values in your data, you have to put placeholders in your dataset and then replace them with a null value.

As an example, the following dataset specifies null values as "###NULL###":

```
<?xml version="1.0" ?>
<dataset>
    <rosters id="1" tig_name="FOO Bat" ibl_team="M\
AD" comments="Test record" status="0" item_type="2\
" />
    <rosters id="2" tig_name="TOR Bautista" ibl_te\
am="MAD" comments="Joey bats!" status="1" item_typ\
e="1" />
    <rosters id="3" tig_name="MAD#1" ibl_team="MAD\
" status="0" comments="###NULL###" item_type="0" /\
>
    <rosters id="4" tig_name="TOR Hartjes" ibl_tea\
m="MAD" comments="Test writer" status="1" item_typ\
e="1" />
</dataset>
```

Once you've loaded the data set, you then need to iterate through it and swap out your token representing null for the real thing.

In our test case's getDataSet() method, we use `PHPUnit_Extensions_Database_DataSet_ReplacementDataSet` to make substitutions. The following example replaces "###NULL###" with a `null` value.

```
<?php
public function getDataSet()
{
    $ds = $this->createFlatXmlDataSet(dirname(__FI\
LE__)
            . '/fixtures/roster-seed.xml');
    $rds = new PHPUnit_Extensions_Database_DataSet\
_ReplacementDataSet($ds);
    $rds->addFullReplacement('###NULL###', null);

    return $rds;
}
```

You can merge data sets if you want to. Here's an example

```
<?php
public function getDataSet()
{
    $mergedDs = PHPUnit_Extensions_Database_DataSe\
t_CompositeDataSet(array());
    $fixturePath = dirname(__FILE__) . DIRECTORY_S\
EPARATOR . 'fixtures/rosters';

    foreach ($fixtures as $fixture) {
        $path =  $fixturePath . DIRECTORY_SEPARATO\
R . "$fixture.xml";
        $ds = $this->createMySQLXMLDataSet($path);
```

```
13          $mergedDs->addDataSet($ds);
14      }
15
16      return $mergedDs;
17  }
```

Using YAML Datasets

Don't like XML? You can always provide datasets in YAML:

```
1  rosters:
2    -
3      id: 1
4      tig_name: "FOO Bat"
5      ibl_team: "MAD"
6      comments: "Test Record"
7      status: 0
8      item_type: 2
9    -
10     id: 2
11     tig_name: "TOR Bautista"
12     ibl_team: "MAD"
13     comments: "Joey bats!"
14     status: 1
15     item_type: 1
16   -
17     id: 3
18     tig_name: "MAD#1"
19     ibl_team: "MAD"
20     comments:
21     status: 0
22     item_type: 0
```

```
  -
    id: 4
    tig_name: "TOR Hartjes"
    ibl_team: "MAD"
    comments: "Test Writer"
    status: 1
    item_type: 1
```

Then, to load that dataset:

```
<?php
public function getDataSet()
{
    return new PHPUnit_Extensions_Database_DataSet\
_YamlDataSet(
        dirname(__FILE__) . '/fixtures/roster-seed\
.yml');
}
```

Using CSV Datasets

CSV (comma-separated value) datasets are also possible:

```
id;tig_name;ibl_team;comments;status;item_type
1;"FOO Bat";"MAD";"Test Record";0;2
2;"TOR Bautista";"MAD";"Joey bats!";1,1
3;"MAD#1";"MAD";null;0;0
4;"TOR Hartjes";"MAD";"Test Writer";1;1
```

You can load that dataset this way:

```
<?php
public function getDataSet()
{
    $dataset = new PHPUnit_Extensions_Database_Dat\
aSet_CsvDataSet();
    $dataset->addTable(
        'rosters', dirname(__FILE__) . '/fixtures/\
roster-seed.csv'
    );
}
```

Array-based Datasets

Sometimes you just want to hand out data as an array, and not mess around with any other file format.

The only catch is that we have to implement our own dataset code...

```
<?php
require_once 'PHPUnit/Util/Filter.php';

require_once 'PHPUnit/Extensions/Database/DataSet/\
AbstractDataSet.php';
require_once 'PHPUnit/Extensions/Database/DataSet/\
DefaultTableIterator.php';
require_once 'PHPUnit/Extensions/Database/DataSet/\
DefaultTable.php';
require_once 'PHPUnit/Extensions/Database/DataSet/\
DefaultTableMetaData.php';

PHPUnit_Util_Filter::addFileToFilter(__FILE__, 'PH\
PUNIT');
```

```
15
16  class Grumpy_DbUnit_ArrayDataSet
17      extends PHPUnit_Extensions_Database_DataSet_Ab\
18  stractDataSet
19  {
20      protected $tables = array();
21
22      public function __construct(array $data)
23      {
24          foreach ($data as $tableName => $rows) {
25              $columns = array();
26
27              if (isset($rows[0])) {
28                  $columns = array_keys($rows[0]);
29              }
30
31              $metaData = new PHPUnit_Extensions_Dat\
32  abase_DataSet_DefaultTableMetaData($tableName, $co\
33  lumns);
34              $table = new PHPUnit_Extensions_Databa\
35  se_DataSet_DefaultTable($metaData);
36
37              foreach ($rows AS $row) {
38                  $table->addRow($row);
39              }
40
41              $this->tables[$tableName] = $table;
42          }
43      }
44
45      protected function createIterator($reverse = F\
46  ALSE)
```

```
47      {
48          return new PHPUnit_Extensions_Database_Dat\
49  aSet_DefaultTableIterator($this->tables, $reverse)\
50  ;
51      }
52
53      public function getTable($tableName)
54      {
55          if (!isset($this->tables[$tableName])) {
56              throw new InvalidArgumentException("$t\
57  ableName is not a table in the current database.")\
58  ;
59          }
60
61          return $this->tables[$tableName];
62      }
63  }
```

Then you implement your `getDataSet()` method;

```
1   <?php
2   public function getDataSet()
3   {
4       $dataset = array(
5           'rosters' => array(
6               array('id' => 1, 'tig_name' => 'Foo Ba\
7   t', 'ibl_team' => 'MAD', 'comments' => 'Test Recor\
8   d', 'status' => 0, 'item_type' => 2),
9               array('id' => 2, 'tig_name' => 'TOR Ba\
10  utista', 'ibl_team' => 'MAD', 'comments' => 'Joey \
11  bats!', 'status' => 1, 'item_type' => 1),
12              array('id' => 3, 'tig_name' => 'MAD#1'\
```

```
13  , 'ibl_team' => 'MAD', 'comments' => 'Draft pick',\
14   'status' => 0, 'item_type' => 0),
15              array('id' => 4, 'tig_name' => 'TOR Ha\
16  rtjes', 'ibl_team' => 'MAD', 'comments' => 'Test W\
17  riter', 'status' => 1, 'item_type' => 1)
18          )
19      );
20
21      return Grumpy_DBUnit_ArrayDataSet($dataset);
22  }
```

In my mind, the only advantage to going through the hassle of creating your own dataset object is that you end up with a dataset that handles missing values a lot easier.

Our First DBUnit Test

How would we write a test for a method that removes a player from a roster? Inside `Roster` we could add this method:

```
1  <?php
2  public function deleteItem($itemId)
3  {
4      $sql = "DELETE FROM teams WHERE id = ?";
5      $sth = $this->db->prepare($sql);
6      return $sth->execute(array($itemId));
7  }
```

The following test verifies that, yes, we can delete items from our rosters table.

```
<?php

public function getConnection()
{
    $dsn = "pgsql:host=127.0.0.1;dbname=ibl_stats;\
user=stats;password=*********";
    $pdo = new PDO($dsn);

    return $this->createDefaultDBConnection($pdo, \
$dsn);
}

public function getDataSet()
{
    // Load your dataset here
}

public function setup()
{
    $this->db - $this->getConnection();
}

public function testRemoveBatterFromRoster()
{
    $testRoster = new Roster($this->db);
    $expectedCount = 3;

    // Database fixture has 4 records in it
    $testRoster->deleteItem(4);
    $rosterItems = $testRoster->getByTeamNickname(\
'MAD');

```

```
33      $this->assertEquals(
34              $expectedCount,
35              count($rosterItems),
36              'Did not delete roster item as expecte\
37  d'
38      );
39  }
```

I think this example shows how powerful DBUnit can be: it provides the infrastructure so that you can test database-driven code as if they were simply units of code instead of waiting for integration tests.

Mocking Database Connections

So we have tests that are talking to the database directly and I have shown you how to use fixtures to create known datasets. It's time to move up to the pure unit test level and make use of mock objects so that we don't have to actually talk to the database any more.

First, create an example of what the database would give us back.

```
1  <?php
2  $databaseResultSet = array(
3      array('tig_name' => 'AAA Foo'),
4      array('tig_name' => 'BBB Bar'),
5      array('tig_name' => 'ZZZ Zazz'));
```

Next, create the mock objects for the PDO object and statement we would be using.

```
<?php
$statement = $this->getMockBuilder('stdClass')
    ->setMethods(array('execute', 'fetchAll'))
    ->getMock();
$statement->expects($this->once())
    ->method('execute')
    ->will($this->returnValue(true));
$statement->expects($this->once())
    ->method('fetchAll')
    ->will($this->returnValue($databaseResultSet));

$db = $this->getMockBuilder('stdClass')
    ->setMethods(array('prepare'))
    ->getMock();
$db->expects($this->once())
    ->method('prepare')
    ->will($this->returnValue($statement));
```

My use of `stdClass` is okay here for the purposes of this particular test. It doesn't really matter what type of object the mocked statement is because we are more interested in what is returned via those two methods. I've used this trick a few times when dealing with mocked objects that need to return other objects.

You can create a mock `PDO` object by extending a `PDO` object and over-riding the constructor. This is necessary because `PDO` contains enough internal dependencies that it cannot be properly serialized, which in turn causes PHPUnit's mock building functions to throw a fatal error.

```
<?php
class MockPDO extends PDO
{
    public function __construct() {}
}

// Then inside your test...
$db = $this->getMockBuilder('MockPDO')
    ->setMethods(array('prepare'))
    ->getMock();
$db->expects($this->once())
    ->method('prepare')
    ->will($this->returnValue($statement));
```

The rest of the test is the same, except we pass in our mocked PDO object instead of the one we created in the test's `setUp()` method.

```
<?php
$roster = new \Grumpy\Roster($db);
$expectedRoster = array('AAA Foo', 'BBB Bar', 'ZZZ\
 Zazz');
$testRoster = $roster->getByTeamNickname('TEST');
$this->assertEquals(
    $expectedRoster,
    $testRoster,
    "Did not get expected roster when passing in k\
nown team nickname"
);
```

Mocking vs. Fixtures

Having now seen the two approaches, when should you use mocks instead of fixtures? In my experience, mocks are the best way to handle things if you are manipulating the results you get back from the database.

If your code is simply returning the results straight from the database, I think unit testing that code is of little value. You are better off investing the time writing functional tests that use fixtures or a database in a known state.

This is also the case if you have chosen to leverage the query language your database uses to act as your "business logic". Let's say you want to have a method that returns the count of players on a roster by using an aggregation function in your SQL.

What should the test look like?

```
<?php
/**
 * @test
 */
public function rosterHasExpectedItemCount()
{
    // Assuming we are using the fixtures from bef\
ore
    $expectedRosterCount = 4;
    $roster = new \Grumpy\Roster($this->db);
    $count = $roster->countItemsByTeamNickname('TE\
ST');

    $this->assertEquals(
```

```
        $expectedRosterCount,
        $count,
        'countItemsByTeamNickname() did not return\
 expected roster count'
    );
}
```

Now to add a method to our Roster class that uses SQL to give us the answer:

```
<?php
/**
 * Return a count of items on a roster when you pa\
ss in the team
 * nickname
 *
 * @param string $nickname
 * @return integer
 */
public function countItemsByTeamNickname($nickname)
{
    $sql = "
        SELECT COUNT(1) AS roster_count
        FROM rosters
        WHERE ibl_team = ?
    ";
    $stmt = $this->db->prepare($sql);
    $stmt->execute(array($nickname));
    $result = $stmt->fetchAll(PDO::FETCH_ASSOC);

    return $result['roster_count'];
}
```

What would be the point of mocking this? It really is what I refer to as a "passthru" function: it's just returning the response of a single action without doing any manipulation to it.

This is not worth doing:

```
<?php
/**
 * @test
 */
public function returnItemCountUsingMockObjects()
{
    $expectedRosterCount = 4;
    $databaseResultSet = array('roster_count' => $\
expectedRosterCount);

    $statement = $this->getMockBuilder('stdClass')
        ->setMethods(array('execute', 'fetchAll'))
        ->getMock();
    $statement->expects($this->once())
        ->method('execute')
        ->will($this->returnValue(true));
    $statement->expects($this->once())
        ->method('fetchAll')
        ->will($this->returnValue($databaseResultS\
et));

    $db = $this->getMockBuilder('stdClass')
        ->disableOriginalConstructor()
        ->setMethods(array('prepare'))
        ->getMock();
    $db->expects($this->once())
```

```
27          ->method('prepare')
28          ->will($this->returnValue($statement));
29
30      $roster = new \Grumpy\Roster($db);
31      $count = $roster->countItemsByTeamNickname('TE\
32  ST');
33
34      $this->assertEquals(
35          $expectedRosterCount,
36          $count,
37          'countItemsByTeamNickname() did not return\
38   expected roster count'
39      );
40  }
```

Having tests are good. Having tests for the sake of writing tests just to use a specific testing tool is useless.

Testing Exceptions

```
<?php
class Foo
{
    protected $api;

    public function __construct($api)
    {
        $this->api = $api;
    }

    public function findAll()
    {
        try {
            $this->api->connect();
            $response = $this->api->getAll();
        } catch (Exception $e) {
            throw new ApiException($e->getMessage(\
))
        }

        return $response;
    }
}
```

If you're into writing what I refer to as "modern PHP", you are definitely going to want to be using exceptions to trap all your non-fatal errors. Code that has exceptions also needs to be tested. Never fear, PHPUnit can show you the way.

Testing Using Annotations

PHPUnit can use annotations to indicate what exceptions and messages it is expecting to encounter when testing code. Let's create a test for our code sample above.

```
<?php
/**
 * Test that makes sure we are correctly triggerin\
g an
 * exception when we cannot connect to our remote \
API
 *
 * @expectedException ApiException
 * @expectedExceptionMessage Cannot connect
 */
public function testThrowsCorrectException()
{
    $api = $this->getMockBuilder('Api')
        ->disableOriginalConstructor()
        ->getMock();
    $api->expects($this->any())
        ->method('connect')
        ->will($this->throwException(new Exception\
('Cannot connect')));
    $foo = new Foo($api);
    $foo->connect();
}
```

PHPUnit is able to verify the exception and corresponding message through the use of two annotations.

`@expectedException` is set to be the exception you are expecting to be thrown while `@expectedExceptionMessage` should be set to the actual message you are expecting to be generated by the exception.

There is a third annotation you can use, `@expectedExceptionCode` if you like to have your exceptions throw messages and an associated code value.

Testing Using `setExpectedException`

You don't have to use annotations if you don't want to. PHPUnit provides a helper method called `setExpectedException()`.

Here's the test re-written using it:

```
1  public function testThrowsCorrectException()
2  {
3      $api = $this->getMockBuilder('Api')
4          ->disableOriginalConstructor()
5          ->getMock();
6      $api->expects($this->any())
7          ->method('connect')
8          ->will($this->throwException(new Exception\
9  ('Cannot connect')));
10     $foo = new Foo($api);
11     $this->setExpectedException('Exception');
12     $foo->connect();
13 }
```

The reason to use `setExpectedException` over annotations is that by placing the call to it right before the code you expect

to generate the exception, you make it easier to debug the test if it fails. When you use the annotation, PHPUnit is simply expecting the test to throw an exception, not caring at what point in the test it happened.

`setExpectedException()` also accepts a second optional parameter that can be used to indicate the message you are expecting the exception to generate.

```
1  public function testThrowsCorrectException()
2  {
3      // Code is the same up to this point...
4      $this->setExpectedException('Exception', 'Cann\
5  ot connect');
6      $foo->connect();
7  }
```

This feature is handy (both with this method and when using annotations) if you have to test code that throws the same exception but with potentially different messages given certain execution paths.

Testing Using `try-catch`

As a third option, you can always trap them using a `try-catch` block in the test itself.

```
/**
 * Test that makes sure we are correctly triggerin\
g an
 * exception when we cannot connect to our remote \
API
 */
<?php
public function testThrowsCorrectException()
{
    try {
        $api = $this->getMockBuilder('Api')
            ->disableOriginalConstructor()
            ->getMock();
        $api->expects($this->any())
            ->method('connect')
            ->will($this->throwException(new Excep\
tion('Cannot connect')));
        $foo = new Foo($api);
        $foo->connect();
    } catch (ApiException $e) {
        return;
    }

    $this->fail('Did not throw expected ApiExcepti\
on');
}
```

If the code under test throws an exception, it will be caught by the `catch` block and the test will pass. Otherwise `fail()` will cause the test to have been considered to not have passed.