

Mockery 0.6 Released - PHP Mock Object Framework

Mockery is a Mock Object framework for PHP, compatible with most unit testing frameworks including PHPUnit. Its purpose is to implement a lightweight grammar for the creation and testing of Mock Objects, Test Stubs, and Test Spies as an alternative to the built-in support offered by PHPUnit, etc.

Mockery is hosted on Github (<http://github.com/padraic/mockery>) where you can find an extensive README covering its API and uses. The Mockery 0.6 release may be installed from the SurviveTheDeepEnd.com PEAR channel at <http://pear.survivedeepend.com>.

Mockery 0.6 features:

- Full Mock Object and Test Stub support
- Lightweight fluent API
- Flexible mocking and stubbing
- Object Interaction Recording
- Natural language syntax and expectation constructs
- Supports generic (untyped) mock objects for rapid prototyping
- Simple partial mocking of real objects
- Both local and global Mock Object call ordering
- Built-in return value queue for repeated method calls
- Support for default expectations
- Support for expectation replacement and stacking
- Fluent API/Law of Demeter mocking

If that sounds complex, it's not! Mockery can be picked up and used with little study.


Why Mockery?

Mockery's objective is to simplify Mock Objects in PHP while maintaining significant flexibility and a default level of intuitive behaviour. In Mockery, Mock Objects behave exactly as you write them with liberal interpretations otherwise applied. Mockery was born out of my own need to innovate the use of Mock Objects in PHP and draw away from the original import of aging Mocking approaches from Java. While Java (and almost every other programming language) has been steadily progressing its mock object libraries, and complementing them with new solutions, PHP has a relatively static approach depending on similarly static library components. That result has seen solutions using clunky APIs, poorly described syntax and behaviour, a lack of focus on the practice of using Mock Objects, user confusion, and raised barriers to new programmers trying to learn about Mock Objects. Mockery is one potential solution to these problems. Also, as a dedicated Test-Driven Design user, I really want something that clicks immediately and doesn't have any gotchas.

Installation

Mockery may be installed from its PEAR channel using:

```
pear channel-discover pear.survivedeepend.com
pear install deepend/Mockery
```

Mockery is written in PHP 5.3 (I know, but all you 5.2 users will get there eventually ). It is released under a New BSD license.

Example

The [README offers a good look](#) at some examples, and explains the API in a lot of detail. If you are trying to figure something out, the README undoubtedly has a section for it. Here's an API example (assuming Mockery namespace used as MK). We're capturing an interaction where we login into a bookmarking service, check for the existence of a "php" tagged bookmark, add three more bookmarks and then recheck if a "php" tag exists (twice for fun). We're mocking the service since we don't actually want to mess with a real account! Following the description closely...

```
$service = \MK::mock('MyService');
$service->shouldReceive('login')->with('user', 'pass')->once()->andReturn(true);
$service->shouldReceive('hasBookmarksTagged')->with('php')->once()->andReturn(false);
$service->shouldReceive('addBookmark')->with('/^http:/', \MK::type('string'))->times(3)->andReturn(
true);
$service->shouldReceive('hasBookmarksTagged')->with('php')->twice()->andReturn(true);
```

The example uses some of the basic parts of Mockery to describe some interaction with a mocked web service class (obviously also stubbing the web service's responses in terms of booleans). The setup is straightforward, easy to follow, and there's zero misinterpretations possible. Our description was likewise simple and uncomplicated. The third line just shows two argument matchers at work, a default regex (intrepreted from any string argument set so long as any eventual string comparison fails and it's a valid regex) and a Type matcher set to match any valid string.

To put this into some perspective, here's an equivalent attempt using PHPUnit in a similar order of thought (excerpt from a test).

```
$service = $this->getMock('MyService');
$service->expects($this->once())->method('login')->with('user', 'pass')->will($this->returnValue(true));
$service->expects($this->once())->method('hasBookmarksTagged')->with('php')->will($this->
returnValue(false));
$service->expects($this->exactly(3))->method('addBookmark')
->with($this->matchesRegularExpression('/^http:/'), $this->isType('string'))
->will($this->returnValue(true));
$service->expects($this->exactly(2))->method('hasBookmarksTagged')->with('php')->will($this->
returnValue(true));
```

Besides the differences in API, there are others. If MyService is just intended as a fake unimplemented object (the class doesn't exist), Mockery carries on and just uses a generic Mock instance without error. PHPUnit will throw an exception, however, stating that login() is not a valid method. If we assume the class is real, but missing some methods, the same thing happens and PHPUnit complains about missing methods. Eventually, you'll get the idea to implement the dependent class... If we add all the relevant methods (say, we mock an interface with all methods declared), PHPUnit STILL fails. This time complaining that hasBookmarksTagged() was expected only once. This occurs because PHPUnit has no capacity for stacking later expectations, and so, it ignores the second (and any later) ones. We can fix that by merging both into a single expectation

using:

```
$service = $this->getMock('MyService');
$service->expects($this->once())->method('login')->with('user', 'pass')->will($this->returnValue(true));
$service->expects($this->exactly(3))->method('hasBookmarksTagged')->with('php')
    ->will($this->onConsecutiveCalls(false, true, true));
$service->expects($this->exactly(3))->method('addBookmark')
    ->with($this->matchesRegularExpression('/^http:/'), $this->isType('string'))
    ->will($this->returnValue(true));
```

Using `onConsecutiveCalls()` to create a return value queue, and merging the two stacked expectations, allows the PHPUnit variant to pass. Unlike Mockery, if there were ten `hasBookmarksTagged()` calls, you would need to add all ten return values (Mockery let's you set the last return value to act infinitely). The merging simply demonstrates that complex class interactions across classes will fall victim to the need to constantly merge expectations until they are unreadable and explain little.

While your mileage may vary, Mockery just doesn't need reworking, deep thought or extra work. Just state what you want your Mock Object to do in plain unconfused English according to your natural thought order! If nothing else, it helps make the expected interaction obvious which makes your tests more readable and explicit.

Feedback

Any issues can be reported via our Github hosted issue tracker. If you wish to discuss Mockery in more detail, you're welcome to join the mailing list at <http://groups.google.com/group/phpmockery>.

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 17:05

Hi Patrick

Just a thought, would it not be super cool to be able to pass closure into `andReturn` so that you could have little if in it or whatever code you want?

Same for methods in general, can i inject closure and replace the method all together? that could be useful as well.

cheers

art Anonymous on May 31 2010, 09:42

Do you have an example? At the last minute, I added a `andReturnUsing()` method which accepts a closure to execute (it's passed any arguments received by the method call). Would that work for you? More than open for any suggestions to make Mockery better.

Anonymous on May 31 2010, 19:24

Sounds like it would do the job :)

To be honest i have not had chance to play with mockery yet so its not something i "need", just using phpunit made me sad sometimes ;). `onConsecutive` call is not as flexible as passing closure would be.

I like to create unit tests that loop over a large set of sample data with expected results so that not only 100% of code gets covered but different edge cases get executed. Passing closure would help sometimes as i could 'grab' result from outer scope or compare something and return respective value from within of mocked method etc.

Thanks for reply : -) and sorry for misspelling your name in first comment :)

Im sure some people will find it useful so thanks for implementing it :)

Cheers! Anonymous on Jun 1 2010, 22:00

I like to create unit tests that loop over a large set of sample data with expected results so that not only 100% of code gets covered but different edge cases get executed. Passing closure would help sometimes as i could 'grab' result from outer scope or compare something and return respective value from within of mocked method etc. Anonymous on Jun 18 2010, 15:00