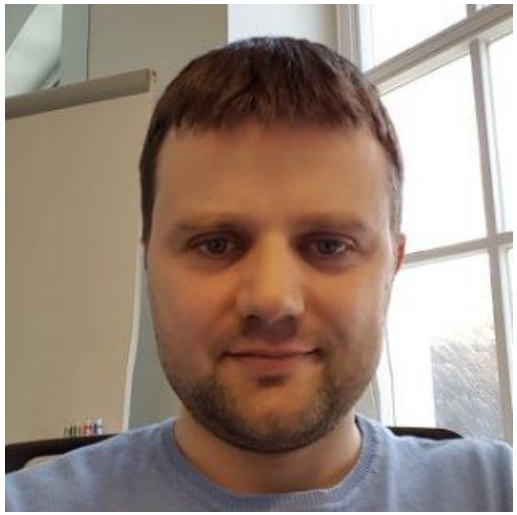
Unit testing with Mockery framework

Linas Gricius / CodeWeek Edition 2016



Linas Gricius

Software Architect / **Intermedix**

https://www.linkedin.com/in/linasgricius

https://github.com/lgricius/mockery-demo

Mockery?

- Mockery is a simple yet flexible PHP mock object framework for use in unit testing with PHPUnit, PHPSpec or any other testing framework.
- Designed as a drop in alternative to PHPUnit's phpunit-mock-objects library, Mockery is easy to integrate with PHPUnit and can operate alongside phpunit-mock-objects without the World ending.

Adding Mockery to the project

```
{
   "name": "lgricius/mockery-demo",
   "require":{
      "php": ">=5.3.2"
   },
   "require-dev": {
      "mockery/mockery": "^0.9.5",
      "phpunit/phpunit": "^5.6"
   }
}
```

PHPUnit Integration

To integrate Mockery with PHPUnit, you just need to define a tearDown() method for your tests containing the following:

```
public function tearDown() {
    \Mockery::close();
}
```

This static call cleans up the Mockery container used by the current test, and run any verification tasks needed for your expectations.

What is a mock object? (1)

In object-oriented programming, mock objects are *simulated objects* that mimic the behavior of real objects in controlled ways.

A programmer typically *creates a mock object to test the behavior of some other object*, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.

What is a mock object? (2)

- Can we mock all the things?
  - Given enough time and effort, yes we can
- Should we mock all the things?
  - Test what we can, mock what we have to
- Should we mock inside our app?
  - Adding mocks shortens the paths we can test

Partial mocks

Partial mocks are useful when you only need to mock several methods of an object leaving the remainder free to respond to calls normally (i.e. as implemented).

Mockery implements three distinct strategies for creating partials. Each has specific advantages and disadvantages so which strategy you use will depend on your own preferences and the source code in need of mocking.

- Traditional Partial Mock
- Passive Partial Mock
- Proxied Partial Mock

Traditional Partial Mock

A traditional partial mock, defines ahead of time which methods of a class are to be mocked and which are to be left unmocked (i.e. callable as normal). The syntax for creating traditional mocks is:

```
$mock = \Mockery::mock('MyClass[foo,bar]');
```

In the above example, the `foo()` and `bar()` methods of MyClass will be mocked but no other MyClass methods are touched. You will need to define expectations for the `foo()` and `bar()` methods to dictate their mocked behaviour.

Passive Partial Mock

A passive partial mock is more of a default state of being.

```
$mock = \Mockery::mock('MyClass')->makePartial();
```

In a passive partial, we assume that all methods will simply defer to the parent class (`MyClass`) original methods unless a method call matches a known expectation. If you have no matching expectation for a specific method call, that call is deferred to the class being mocked. Since the division between mocked and unmocked calls depends entirely on the expectations you define, there is no need to define which methods to mock in advance.

Proxied Partial Mock

A proxied partial mock is a partial of last resort. You may encounter a class which is simply not capable of being mocked because it has been marked as final. Similarly, you may find a class with methods marked as final. In such a scenario, we cannot simply extend the class and override methods to mock - we need to get creative.

```
$mock = \Mockery::mock(new MyClass);
```

Mock Expectations (1)

Once you have created a mock object, you'll often want to start defining how exactly it should behave (and how it should be called). This is where the Mockery expectation declarations take over.

```
shouldReceive(method_name)
with(arg1, arg2, ...) / withArgs(array(arg1, arg2, ...))
once()
andReturn(value1, value2, ...)
andThrow(Exception)
```

Mock Expectations (2)

Tells the expectation to bypass a return queue and instead call the real method of the class that was mocked and return the result:

```
passthru()
```

Returns the current mock object from an expectation chain. Useful where you prefer to keep mock setups as a single statement, e.g.

```
$mock = \Mockery::mock('foo')
          ->shouldReceive('foo')
          ->andReturn(1)
        ->getMock();
```

Mocking Public Static Methods

Static methods are not called on real objects, so normal mock objects can't mock them.

Mockery supports class aliased mocks, mocks representing a class name which would normally be loaded (via autoloading or a require statement) in the system under test.

```
$mock = \Mockery::mock('alias:\MyStaticClass');
```

These aliases block that loading (unless via a require statement - so please use autoloading!) and allow Mockery to intercept static method calls and add expectations for them.

Mocking Final Classes

In a compromise between mocking functionality and type safety, Mockery does allow creating “proxy mocks” from classes marked final, or from classes with methods marked final.

This offers all the usual mock object goodness but the resulting mock will not inherit the class type of the object being mocked, i.e. it will not pass any instanceof comparison.

Notes (1)

```
/**
* Allows additional methods to be mocked that do not explicitly exist on mocked class
* @param String $method name of the method to be mocked
*/
public function shouldAllowMockingMethod($method);

/**
* Set mock to ignore unexpected methods and return Undefined class
* @param mixed $returnValue the default return value for calls to missing functions on this mock
* @return Mock
*/
public function shouldIgnoreMissing($returnValue = null);
```

Notes (2)

```
/**
* @return Mock
*/
public function shouldAllowMockingProtectedMethods();


/**
* Set mock to defer unexpected methods to its parent if possible
*
* @return Mock
*/
public function shouldDeferMissing();
```

References

- http://docs.mockery.io/en/latest/
- https://speakerdeck.com/ramsey/mocking-with-mockery-midwest-php-2016
- https://www.sitepoint.com/mock-test-dependencies-mockery/
- https://speakerdeck.com/mfrost503/mocking-dependencies-in-phpunit
- https://speakerdeck.com/stuartherbert/when-to-mock