DZone Refcardz

CONTENTS INCLUDE:

PHPUnit: PHP Test-Driven Development

Automated Tools to Improve Your PHP Code Quality

By Giorgio Sironi

HOW TO WRITE A TEST

Extensively testing a web application is the only way to make sure that the code you have written really works. Formal proofs of correctness are impossible or impractical to put together for the majority of computer programs. Therefore, actually running code in a sandbox under controlled conditions is the way to uncover bugs and drive the development of new features.

Of course, manual testing cannot be totally substituted by automated approaches, but automating a large part of an application's tests leads to greater testing frequency, and quicker discovery of issues and regression.

Automated testing is based on the concept of test cases (classes in PHPUnit's case) that compose a complete test suite. This suite, or a subset of it, can be run at will to check the production code correctness.

In this approach, the simplest and most effective way to write tests is to write code—simple code, but with the mandatory expressiveness of imperative languages that should give you the needed freedom.

Hot Tip

Besides the quality assurance side of testing, there is also the advantage in feedback that a good test suite provides. The more fine-grained the tests are, the more internal quality is put under "the lens". Clean code is easy to test, while you can't get away with technical debt if you have to write automated tests at the same time.

Tests are the first users of your code. They will tell you much about its simplicity of use, the isolation from component dependencies and the side effects that may arise.

Installation

In the first part of this DZone Refcard, we'll test array_sum(), a simple, native PHP function that computes the sum of values in an array. The purpose of this first test is to introduce the mechanics of PHPUnit usage.

Before writing a test at all, we need PHPUnit. You can simply grab it via its PEAR channel:

```
sudo pear channel-discover pear.phpunit.de
sudo pear channel-discover ... # other channels
sudo pear install phpunit/PHPUnit
sudo pecl install xdebug # for code coverage
```

The other channels where PHPUnit pulls components from may change in the future, so refer to the official documentation (http://phpunit.de/manual). This DZone Refcard is updated to the 3.5 version.

The first test

Here is how a test case looks like:

```
<?php
class ArraySumTest extends PHPUnit_Framework_TestCase
{
    public function testSumTheValuesOfAnEmptyArray()
    {
        $sum = array_sum(array());
        $this->assertEquals(0, $sum, "The sum of an empty array is computed
as $sum.");
    }
}
```

Essentially, this is a class that extends PHPUnit_Framework_TestCase so that it can be run by the /usr/bin/phpunit script. Here is a sample output:

```
[08:08:01][giorgio@Desmond:~/txt/phpunit_refcard]$ phpunit --colors howtowriteat
est1.php
PHPUnit 3.5.0 by Sebastian Bergmann.

.

Time: 0 seconds, Memory: 3.50Mb

OK (1 test, 1 assertion)
```

The methods whose names start with 'test' are executed, one at a time, on a different instance of this Test Case. You can execute whichever code you prefer in order to produce a set of results to confront with the ones you expect.

You can use methods on the Test Case that start with 'assert' to execute checks on the result of your computations that will ensure the system is behaving correctly. PHPUnit provides many simple assertion methods, but you can always define your own by adding private methods.

Executing phpunit filename.php from the command line would run the test. You can also run all the tests in a certain folder (subfolders included) simply by passing a path to the directory.

This was the simplest test we could possibly write: it will provide good practice to start with and you can augment the complexity as you gain confidence about the System Under Test (SUT).

Now that we are capable of running a test on your machine, let's gain confidence and expand the test a bit:

```
<?php
class ArraySumTest extends PHPUnit_Framework_TestCase
{
    public function testSumTheValuesOfAnEmptyArray()
    {
        $sum = array_sum(array());
        $this->assertEquals(0, $sum);
    }

    public function testSumTheValuesOfAnArrayWithOneValue()
    {
        $sum = array_sum(array(42));
        $this->assertEquals(42, $sum);
    }

    public function testSumTheValuesOfAnArrayWithManyElements()
    {
        $sum = array_sum(array(1, 2, 3, 4, 5, 6));
        $this->assertEquals(21, $sum);
    }

    public function testSumTheValuesOfAnArrayWithFloatValues()
    {
        $sum = array_sum(array(1, 2.5));
        $this->assertEquals(3.5, $sum);
    }
}
```

A failing test

Here is a failing test instead. We expect that array_sum works recursively by summing up all the values in internal arrays:

```
<?php
class ArraySumTest extends PHPUnit_Framework_TestCase
{
    // ...
    public function testSumsRecursively()
    {
        $sum = array_sum(array(1, array(2, 3)));
        $this->assertEquals(6, $sum);
    }

}
```

When run, this test gives the following result:

```
[08:06:50][giorgio@Desmond:~/txt/phpunit_refcard]$ phpunit --colors howtowriteat
est3.php
PHPUnit 3.5.0 by Sebastian Bergmann.

....F

Time: 0 seconds, Memory: 3.50Mb

There was 1 failure:

1) ArraySumTest::testSumsRecursively
Failed asserting that <integer:1> matches expected <integer:6>.

/home/giorgio/Dropbox/txt/phpunit_refcard/howtowriteatest3.php:31

FAILURES!
Tests: 5, Assertions: 5, Failures: 1.
```

We were asserting that the production code performed a task that it couldn't. The test tells us it is not living up to our expectations.

Before diving into PHPUnit's features, I want to highlight some methods that complement assertions in the flow control of PHPUnit.

Method	Effect
$this->fail($message = '')	Will instantly make the test fail, like in the case an assertion fails.
$this->markTestSkipped()	Tells PHPUnit that this test should be interrupted and marks it with an (S) in the results.
$this->markTestIncomplete()	Tells PHPUnit that this test is incomplete and shouldn't be run either. It will be marked with an (i) in the results and will turn the green bar of PHPUnit into an orange one.

ASSERTIONS

Assertion methods are used to check the result of your test's execution with predefined expected values. All these methods are available on $this, as they are defined on PHPUnit_Framework_TestCase. I report here the ones that, by experience, are the most handy and widely called in my test suites.

Basic assertions are the ones you always see in introductory material, and can serve many needs. They are also versatile, since corner cases can be reconduced to assertTrue() and assertEquals() in nearly any scenario.

Method	Effect
assertEquals()	Fails if the two arguments are not equal (checks with the == operator).
assertSame()	Fails if the two arguments are not identical (checks with ===), that is, they are not the same object or they are not scalar equal in value and type.
assertTrue()	Fails when this argument is not true.
assertFalse()	Fails when this argument is not false.
assertEmpty()	Fails when empty() on the parameter is false.
assertNull(), assertNotNull()	Checks if the variable is the special value NULL or not.
assertInstanceOf()	Checks that an object is an instance of (using this operator) the class or interface passed.
assertInternalType()	Checks if a scalar is of a particular PHP type-like integer or boolean.

Here is a first sample of these methods in action:

```
<?php
class BasicAssertionsTest extends PHPUnit_Framework_TestCase
{
    public function testBasicAssertionsWorkAsExpected()
    {
        $this->assertEquals(1, 1);
        $this->assertEquals(1, "1");
        $this->assertEquals(3.0001, 3.000, 'Floats should be compared with
a tolerance', 0.001);
        $this->assertSame(42, (int) "42");
        $this->assertTrue(true);
        $this->assertFalse(false);
        $this->assertEmpty('0');
        $this->assertNull(null);
        $this->assertNotNull(42);
        $this->assertInstanceOf('stdClass', new stdClass);
        $this->assertInternalType('string', 'hello, world');
    }
}
```

There are more elaborate assertions which are indeed frequently called:

Method	Effect
assertArrayHasKey(), assertArrayNotHasKey()	Checks the presence of a key in an array.
assertContains($needle, $haystack)	Checks the presence of a value in an array, both numerical or associative. Works also with two strings, checking that the first is contained in the other one.
assertContainsOnly($type)	Checks that an array is composed only of $type values, like integers or booleans.
assertRegExp()	Checks that a string matches a PREG-based regular expression.

Here is a sample test case:

```
<?php
class ArrayAndStringsAssertionsTest extends PHPUnit_Framework_TestCase
{
    public function testArrayAndStringsAssertionsWorkAsExpected()
    {
        $this->assertArrayHasKey('key', array('key' => 'value',
'anotherKey' => 'anotherValue'));
        $this->assertArrayNotHasKey('key', array('anotherKey',
'yetAnotherKey'));
        $this->assertContains('value', array('value'));
        // works for strings too
        $this->assertContains('world', 'Hello, world!');
        // this would work with any primitive type
        $this->assertContainsOnly('string', array('value', 'otherValue'));
        // PCRE regular expression
        $this->assertRegExp('/[A-Za-z ,!]+/', 'Hello, world!');
    }
}
```

There are also special cases:

Method	Effect
assertFileExists()	Check that a file is present in the filesystem based on its path.
assertFileEquals()	Check that the content of two files are equal.
assertGreatherThan(), assertGreatherThanOrEqual(), assertLessThan(), assertLessThanOrEqual()	All assertion methods oriented to operate with >, >=, < and <= over the two values passed.

Here is an example of them in action:

```
<?php
class ExoticAssertionsTest extends PHPUnit_Framework_TestCase
{
    public function testExoticAssertionsWorkAsExpected()
    {
        $this->assertFileExists(__FILE__);
        $this->assertFileEquals(__FILE__, __FILE__);
        $this->assertGreaterThan(23, 42);
        $this->assertGreaterThanOrEqual(42, 42);
        $this->assertLessThan(16, 15);
        $this->assertLessThanOrEqual(16, 16);
    }
}
```

Custom assertions are great tools (you just have to define an assert*() method). However, taking advantage of the assertions already supported by the framework will not require that you reinvent the wheel, or extend your base Test Case class in all your tests.

All assertion methods take a last optional argument $message, which is displayed when they fail. You may enhance your tests error messages when they are not clear with this additional parameter.

FIXTURES

Fixtures are a common way to set up objects that are used in all test methods before each test run.

In PHPUnit, fixtures can be implemented with the setUp() and tearDown() hook methods, which can populate $this private fields, preparing them for usage by the single test methods.

There are also two static methods, setUpBeforeClass() and tearDownAfterClass(), that are called only one time for each test case class. Because of their static modifier, they can only access static properties and are typically used for resources that are particularly heavy to set up, like database connections.

Hot Tip PHPUnit creates a new Test Case object for each run, so you won't have to reset your fields between the test methods. Therefore, each test method is run in isolation and won't interfere with the $this fields set in previous runs.

Due to this isolation property, tearDown() is usually not present unless there is some external resource to release, like a database. For in-memory objects, garbage collection will be enough.

Due to the space available, this Refcard won't go into mechanisms for setting up global variables or static attributes, which are usually more a design problem than a testing one.

Here is a test case which uses fixtures implemented with setUp().

```
<?php
class FixtureTest extends PHPUnit_Framework_TestCase
{
    private $systemUnderTest;

    public function setUp()
    {
        $this->systemUnderTest = new stdClass;
    }

    public function testFixtureWorkAsExpected()
    {
        $this->assertInstanceof('stdClass', $this->systemUnderTest);
        $this->systemUnderTest->field = true;
    }
    public function testAnotherTestIsRunInIsolation()
    {
        $this->assertFalse(isset($this->systemUnderTest->field));
    }

    public function tearDown()
    {
        // don't need to do anything if garbage collection would take care
of this
        // close your database connections, for example
    }
}
```

ANNOTATIONS

PHPUnit supports the use of some annotations on test case and single test methods with the goal of transforming into declarative options some common behaviors diffused in test suites. For example, running the same test multiple times with a different set of input data is an operation common enough to warrant its own annotation. The mechanism of these multiple runs isn't duplicated throughout the test suite of our projects, instead it is kept in the framework.

PHP does not have native support for annotations, so they are embedded into the docblocks of classes and methods, much like the @param and @return annotations used for API documentation. These annotations will save you from writing boilerplate code you would otherwise end up repeating in a bunch of tests that all look alike, or inheriting from some base test case class that gets longer and longer.

@dataProvider

This annotation tells PHPUnit to run a single test method multiple times by passing a different data set as method parameters each time.

PHPUnit will display a different failure for each of the different runs, so that multiple tries are independent and one failing does not affect the sebsequent execution of the others. Running the same test code with a foreach() cycle would not have the same effect.

```
[08:53:48][giorgio@Desmond:~/txt/phpunit_refcard]$ phpunit --colors --filter Sum
OperatorsWorks annotations.php
PHPUnit 3.5.0 by Sebastian Bergmann.

..F

Time: 0 seconds, Memory: 3.50Mb

There was 1 failure:

1) AnnotationsTest::testSumOperatorsWorksCorrectly with data set #2 (10, 12, 42)
Failed asserting that <integer:22> matches expected <integer:42>.

/home/giorgio/Dropbox/txt/phpunit_refcard/annotations.php:22

FAILURES!
Tests: 3, Assertions: 3, Failures: 1.
```

@depends

Hot Tip This annotation declares a test as dependent on a previous one, saying essentially that when a basic test on the SUT fails, it is not worthy to even try to run a more complex, related one.

In case the SUT experiences a large regression, the errors presented by a test run would be much more focused, consisting only of the tests that failed first and excluding their dependent siblings.

Defining dependencies is usually only a good idea inside the same test case class. In fact, it is the only scale supported by this feature. The dependent test will be marked as 'skipped' when the first one fails, otherwise it will be run normally. Moreover, the first test can pass a computation result to its dependency by returning it (the dependent will accept it as a parameter). A classic example is testing the add() and remove() methods of a collection:

The first test will exercise add(), check that the inserted element is present and return the collection.

The second test will exercise remove() on the passed collection, and check that the element is not present anymore.

```
[08:56:09][giorgio@Desmond:~/txt/phpunit_refcard]$ phpunit --colors --filter Dep
end annotations.php
PHPUnit 3.5.0 by Sebastian Bergmann.

FS

Time: 0 seconds, Memory: 3.50Mb

There was 1 failure:

1) AnnotationsTest::testFailsAndAnotherTestDependsOnIt
I fail in order to stop my dependent test from running.

/home/giorgio/Dropbox/txt/phpunit_refcard/annotations.php:27

FAILURES!
Tests: 1, Assertions: 0, Failures: 1, Skipped: 1.
```

@expectedException (with @expectedExceptionCode and @expectedExceptionMessage)

This annotation asserts that the code contained in the test methods throws an exception, and optionally checks its code and its message.

Note that any line of code can be throwing the exception, as long as it is called inside the test method. If you want a more accurate check on which statement should raise an exception, stick to try/catch blocks.

@group

Hot Tip Grouping aids you in running the exact amount of tests during development, and not the entire suite, which can require much more time. You will get feedback on a more frequent basis, and only from the code you select. This leaves the rest of the code base for subsequent integration (pre-commit or pre-push).

This annotation is not applicable to test methods, but only on test cases. It defines a test case as pertaining to a particular group that can be used as a filter to run all the grouped tests independently from the rest of the suite.

A test case can be in multiple groups. They can be thought of as labels rather than folders (which are an independent filter.)

For examples, you may use annotations such as @group acceptance (for Selenium-based tests); @group functional (for tests which do not exercise a single unit and, hence, are a bit slower to run), and @group ticket-42 (for the tests regarding a particular bug fix).

The command used to filter tests of a particular group is *phpunit --group*, which will be treated later in the command line section. However, you can also filter groups in the XML configuration.

@runTestsInSeparateProcesses

This too is a test case annotation. It is not diffused and should be used only for special case tests. It runs tests in different executions of the *php* command, so that they do not affect each other. This mechanism can be useful when testing something related to autoloading or a similar necessary global state. For example, you may be testing that the autoloader loads the same class multiple times under different conditions, and this can only be done in different processes.

```
<?php
/**
 * @group acceptance
 * @group someOtherGroup
 */
class AnnotationsTest extends PHPUnit_Framework_TestCase
{
    public static function dataToFeed()
    {
        return array(
            array(1, 2, 3),
            array(4, 2, 6),
            array(10, 12, 22)
        );
    }

    /**
     * @dataProvider dataToFeed
     */
    public function testSumOperatorsWorksCorrectly($a, $b, $total)
    {
        $this->assertEquals($total, $a + $b);
    }

    public function testFails()
    {
        $this->fail('I fail in order to stop my dependent test from
running.');
    }

    /**
     * @depends testFails
     */
    public function testIAmDependent()
    {
        $this->fail('I would fail too, but I\'m not run.');
    }

    /**
     * @expectedException Exception
     * @expectedExceptionCode 400
     * @expectedExceptionMessage This is the message
     */
    public function testRaisesException()
    {
        throw new Exception('This is the message', 400);
    }
}
```

Test Doubles

Test Doubles are substitutes for real objects that you don't want to involve in your tests. By injecting Test Doubles in your System Under Test, you can effectively isolate it from the rest of the system and let it call its collaborators without null checks. You can also define what the Test Doubles expect as method parameters, or what they should return.

The taxonomy of Test Doubles comprehends:

Pattern	Meaning
Dummies	Objects that exist only to satisfy type hints and runtime checks. They are passed around but no methods are called on them.
Stubs	Objects that return canned results when their methods are called.
Mocks	Stubs that can also check what parameters are passed to them.
Fakes	Real implementations of collaborators, but much more lightweight than the real object.

PHPUnit offers support for automatic generaton and instantiation of Dummies, Stubs, and Mocks. It subclasses the defined class, overrides methods and eval() the resulting code. In PHPUnit, Mock is used as an umbrella term covering any Test Double category.

The getMock() or getMockBuilder() methods can be called to obtain the mock, or a Builder implementation over the mock itself. They accept an interface or a concrete class as the first argument. getMockForAbstractClass is the equivalent of getMock() for this special case.

Hot Tip I contributed getMockBuilder() to provide a cleaner way for complex mock instantiation, which usually requires calls with 7 arguments to getMock(). It provides a fluent interface with different methods to set the creation options one by one, prior to creating the mock.

Once you have a mock, you can perform the expects() call on it to create an expectation object. The expectation object then presents different methods which provide a fluent interface:

Method	Effect
method()	Defines the name of the method it refers to.
with()	Specifies assertions to make on the parameters passed. In the simplest cases, you call it with the value you would use to call the method, in the identical order.
will()	Defines the behavior of the overridden method as traits such as what to return or whether to throw an exception.

Here is a test case which covers many instances of stubs and mocks usage:

```
<?php
class TestDoublesTest extends PHPUnit_Framework_TestCase
{
    public function testInstantiatesADummy()
    {
        $dummy = $this->getMock('stdClass');
        $this->assertInstanceOf('stdClass', $dummy);
    }

    public function testInstantiatesAStub()
    {
        $stackStub = $this->getMock('SplStack');
        $stackStub->expects($this->any())
                  ->method('pop')
                  ->will($this->returnValue(42));

        $this->assertEquals(42, $stackStub->pop());
    }

    public function testSetsUpAStubMethodWithACallback()
    {
        $callback = function($argument) {
            $map = array(
                'key' => 'value',
                'otherKey' => 'otherValue'
            );

            return $map[$argument];
        };

        $stackStub = $this->getMock('SplStack');

        // don't ask me why a Stack has getter and setters
        // I use it only because of its availability
        $stackStub->expects($this->any())
                  ->method('offsetGet')
                  ->will($this->returnCallback($callback));
        $this->assertEquals('value', $stackStub->offsetGet('key'));
    }

    /**
     * @expectedException InvalidArgumentException
     */
    public function testThrowsAnException()
    {
        $stackStub = $this->getMock('SplStack');
        $stackStub->expects($this->any())
                  ->method('push')
                  ->will($this->throwException(new
InvalidArgumentException));

        $stackStub->push(42);
    }

    public function testInstantiatesAMockAndPutExpectationOnAParameter()
    {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())
              ->method('push')
              ->with(42);

        $stackMock->push(42);
    }

    public function
testInstantiatesAMockAndPutExpectationOnMultipleParameters()
    {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())
                  ->method('offsetSet')
                  ->with(2, 42);

        $stackMock->offsetSet(2, 42);
    }

    public function
testInstantiatesAMockAndPutExpectationsOnParametersDifferentFromEqualTo()
    {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())
                  ->method('offsetSet')
                  ->with($this->anything(), $this->identicalTo('42'));

        $stackMock->offsetSet(2, '42');
    }

    public function
testInstantiatesAMockAndPutOtherExpectationsOnParameters()
    {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())
                  ->method('offsetSet')
                  ->with($this->isType('int'), $this-
>isInstanceOf('stdClass'));

        $stackMock->offsetSet(2, new \stdClass);
    }

    public function
testInstantiatesAMockAndPutYetOtherExpectationsOnParameters()
    {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())
                  ->method('offsetSet')
                  ->with($this->lessThan(3), $this->isTrue());

        $stackMock->offsetSet(2, true);
    }

    public function testCallsAMockTwice()
    {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->exactly(2))
                  ->method('push');

        $stackMock->push(23);
        $stackMock->push(42);
    }

    public function testShouldNotCallAMock()
    {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->never())
                  ->method('push');

        // this would fail

        // $stackMock->push(42);
    }

    /**
     * Each call on the MockBuilder apart from getMock() is optional.
     */
    public function testInstantiatesAMockUsingBuilder()
    {
        $arrayIteratorMock = $this->getMockBuilder('ArrayIterator')
                                  ->setMethods(array('current', 'next'))
                                  ->disableOriginalConstructor()
                                  ->disableOriginalClone()
                                  ->getMock();
    }
}
```

COMMAND LINE USAGE

The phpunit script is the main means to run PHPUnit-based test suites, even if there are ways to bypass it like Phing (http://phing.info/trac/) tasks for purposes of Continuous Integration.

Knowing how to use *phpunit* effectively is therefore crucial to leverage your test suite power. These are the various options available to modify its behavior:

Switch	Effect
--configuration <file>, --no-configuration	Includes a phpunit.xml file different from the one in the working directory or excludes its automatic recognition.
--coverage-html <directory>	Generates an HTML report on code coverage in the specified folder. The xdebug extension is required for code coverage to be collected and the test execution will be much slower.
--colors	Specifies to use colors in the output as a nice way to visualize the test success.
--bootstrap <file>	Defines a PHP file to include before starting the test suite execution. It is included also in the configuration from some releases.
--filter	Lets you filter test methods that match the passed pattern. You won't have to comment on the other tests when you want to focus on a single one.
--group	Filters the tests defined by the current suite and runs only the ones in the specified group.
--exclude-group	Runs all the tests except the ones in the chosen group.
--list-groups	Shows all the groups available for running independently.
--include-path	Sets a particular include_path ini directive for running the test.
-d key=value	Sets a php.ini value (temporarily).
--verbose	Displays a list of test case names as the execution goes along, so that you can catch errors during long run just when they happen, instead of waiting for the whole test suite to finish.
--version	Displays current version of PHPUnit. Handy to know, since running it without parameters starts the whole test suite. -v is not supported.
--testdox	Generates a "kind" of Agile documentation from the test names. If you write test names well, it documents what your class or component under test does for a living.

CONFIGURATION

PHPUnit supports configuration of the test suite via an XML file that can be stored under version control. Common options like bootstrap files, logging formats or filters can be saved in this file instead of being passed with each PHPUnit command.

PHPUnit looks for phpunit.xml and phpunit.xml.dist, which are assumed to be present in the working directory. Usually one of them is kept in the test suite root directory.

The common practice is to store phpunit.xml dist in the source control system and ignore phpunit.xml, so that users can personalize their run if they do not have the tools for running all tests available (such as Selenium or staging http servers).

It can also be the case that you do not want to run slow integration tests sometimes. With a phpunit.xml.dist present however, you will be able to run tests just after a checkout.

Root element

<phpunit> is the root element. It accepts various attributes such as "bootstrap", which specifies a boostrap file to execute prior to the tests, or "colors", which prescribes the use of colors like red and green in the output.

Test suites

The <testsuites> and <testsuite> elements let you group tests from different folders. They can contain these two elements:

- <file>, which defines a single PHP file.
- <directory>, which will include all the tests inside that directory or a subdirectory. <directory> accepts a suffix attribute to filter the test case name: by default it is 'Test'.

Filtering, logging, and more

Inside the configuration, you also have different modifiers available. You can:

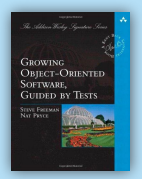
- Filter tests of a certain <group> to run (or not run) them.
- Define code coverage inclusion (and exclusion) of files. The <filter> element refers to code coverage.
- Log results and statistics in various formats, with the <logging> element. It may contain various <log> entries, of various types: coverage-html, coverage-xml, json, tap, junit, testdox-html, testdox-text.
- Attach listeners with the <listeners> and <listener> elements.
- Configure a Selenium RC server to run acceptance tests, with the <selenium> and <browser> elements.

```
<phpunit bootstrap="/path/to/bootstrap.php"
         colors="false">
    <testsuites>
        <testsuite>
            <directory>tests/</directory>
            <file>DoctrineTest.php</file>
        </testsuite>
    </testsuites>

    <groups>
        <include>
            <group>functional</group>
        </include>
        <exclude>
            <group>acceptance</group>
        </exclude>
    </groups>

    <filter>
        <blacklist>
            <directory suffix=".php">library/</directory>
             <exclude>
                 <file>library/App</file>
            </exclude>
        </blacklist>
        <whitelist>
            <directory suffix=".php">application/</directory>
            <file>bootstrap.php</file>
            <exclude>
                <directory suffix=".php">application/views</directory>
            </exclude>
        </whitelist>
    </filter>

    <logging>
        <log type="coverage-html" target="/tmp/coverage-report-folder" />
        <log type="junit" target="/tmp/junit-log.xml" />
        <log type="testdox-html" target="/tmp/testdox.html"/>
    </logging>

    <listeners>
        <listener class="App_Test_FunctionalTestListener" file="library/App/
Test/FunctionalTestsListener.php">
            <arguments>
                <string>staging</string>
            </arguments>
        </listener>
    </listeners>

    <selenium>
        <browser name="Ubuntu-Firefox"
                 browser="*firefox /usr/bin/firefox"
                 host="localhost"
                 port="4444" />
    </selenium>
</phpunit>
```

ABOUT THE AUTHOR

Giorgio Sironi is a Bachelor of Computer Engineering and works as a PHP software architect while continuing his studies in a Master Program at Politecnico di Milano. During the last six years he transitioned from his early work as a websites developer to the creation of web-based applications. He practices Test-Driven Development every day and believes in testing as a design tool. Giorgio is an international speaker at PHP conferences and you may have already heard of him as the author of the *Practical PHP Testing* free ebook, or the "Practical PHP Patterns" series on DZone.

RECOMMENDED BOOK



Test-Driven Development (TDD) is now an established technique for delivering better software faster. TDD is based on a simple idea: Write tests for your code before you write the code itself. However, this "simple" idea takes skill and judgment to do well. Now there's a practical guide to TDD that takes you beyond the basic concepts. Drawing on a decade of experience building real-world systems, two TDD pioneers show how to let tests guide your development and "grow" software that is coherent, reliable, and maintainable.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-04-2
ISBN-10: 1-936502-04-6
50795
9 781936 502042
$7.95

Version 1.0