Demeanor Documentation

Release 0.2

Christopher Davis

June 08, 2014

Contents

Demeanor is PHP testing framework that supports several different test formats.

1. Specification Tests (SpecBDD)
2. Unit Tests
3. PHPT Tests

CHAPTER 1

Contents

1.1 Configuration

Demeanor use JSON for configuration and looks for the configuration files `demeanor.json` and `demeanor.dist.json` by default. A custom configuration file can be used by using the `-c` or `--config` command line options.

```
./vendor/bin/demeanor --config a_custom_config.json
```

1.1.1 Test Suites

The centerpiece of Demeanor's configuration is the `testsuites` argument. If no suites are defined, the command line test runner will error. Similarly if `testsuites` isn't a JSON object, the runner will error.

Each test suite can have it's own bootstrap file(s) as well as define it's own test locations.

Here's a complete example:

```
{
    "testsuites": {
        "A Test Suite": {
            "type": "unit",
            "bootstrap": [
                "test/bootstrap.php"
            ],
            "directories": [
                "test/classes",
                "test/another_directory"
            ],
            "files": [
                "test/path/to/a/file.php"
            ],
            "glob": [
                "test/files/*Test.php"
            ],
            "exclude": {
                "directories": [
                    "test/classes/not_this_one"
                ]
                , "files": [
                    "test/path/to/exclude.php"
                ]
```

```
                , "glob": [
                    "test/exclude/*.php"
                ]
            }
        }
    }
}
```

- `type` is the type of test suite. This just tells demeanor what to do with the suite itself. Valid values are `unit`, `spec`, and (eventually) `story`.
- `bootstrap` is a list of files that will be `require_once`'d before of the suites tests are run. Use these files to do any setup for the test suite.
- `directories` tells demeanor to look for files in a directory. What files it looks for depends on the suite type.
  - `unit` test suites look for files that end in `Test.php`
  - `spec` test suites look for files that end in `spec.php`
- `files` is a list of files that will be treated as if they contain test cases.
- `glob` is a list of glob patterns that will be used to locate test files.
- `exclude` is used to blacklist files from your test suite. It's an object that looks very similar to the test suite itself. `directories`, `files`, and `glob` work exactly as they do in the test suite itself.

1.1.2 Default Test Suites

There's a good chance you won't want to run all your test suites all the time. For instance, acceptance tests often take a long time – they'll test your complete system end to end.

That's where the `default-suites` configuration option comes in. When defined only the test suites defined in it's array (or string) will be run with the naked `demeanor` command.

`default-suites` may be an array.

```
{
    "default-suites": ["a_suite"],
    "testsuites": {
        "a_suite": {
            "type": "spec",
            "directories": [
                "test/spec"
            ]
        }
    }
}
```

Or it can just be a string.

```
{
    "default-suites": "a_suite",
    "testsuites": {
        "a_suite": {
            "type": "spec",
            "directories": [
                "test/spec"
            ]
        }
```

```
    }
}
```

If a suite that doesn't exist is supplied, the `demeanor` CLI will fail.

```
{
    "default-suites": "this-will-not-work",
    "testsuites": {
        "a_suite": {
            "type": "spec",
            "directories": [
                "test/spec"
            ]
        }
    }
}
```

How can I run other test suites then?

Use the `--testuite` (or `-s`) command line option.

```
./vendor/bin/demeanor --testsuite a_suite
```

Or use a few of them.

```
./vendor/bin/demeanor -s a_suite -s another_suite
```

Or use the `--all` (or `-a`) option to run all test suites.

```
./vendor/bin/demeanor --all
```

1.1.3 Subscribers

`subscribers` can be defined in `demeanor.json` to add event subscribers to that hook in and change how the test runs work.

`subscribers` should be a list of class names that implement `Demeanor\Event\Subscriber`.

```
{
    "subscribers": [
        "Acme\\Example\\TestSubscriber"
    ],
    "testsuites": {
        ...
    }
}
```

These subscribers should have a argumentless constructor. Demeanor uses the event subscriber API itself, look in the `src/Subscriber` directory of the demeanor repo for examples.

1.2 The Test Context

Demeanor has a concept of a test context that's defined by the `Demeanor\TestContext` interface. This context object is **always** the first argument to any test method or callback.

The test context simply lets you pass messages back to the test case – telling it to fail, skip, expect and exception, or log a message. Additionally the context implements `ArrayAccess` so you can use it to store values for the duration of the test run.

Here's an example for a unit test.

```
<?php
// SomeTest.php

use Demeanor\TestContext;

class SomeTest
{
    public function testStuff(TestContext $ctx)
    {
        $ctx->skip('some reason why the test is to be skipped');
    }

    public function testThisWillBeFailed(TestContext $ctx)
    {
        $ctx->fail('The reason for the failure here');
    }

    public function testLogs(TestContext $ctx)
    {
        $ctx->log('some message here');
    }

    public function testExpectException(TestContext $ctx)
    {
        $ctx->expectException('Exception');
        throw new \Exception('this does not cause the test to fail');
    }
}
```

Of these `log` is probably the more interesting. These message are not shown to the user unless they've cranked up the verbosity on the command line test runner (with the `-v|-vv|-vvv` options) or the test fails.

The same test context object that's passed to the actually test method/callback is also passed to the before and after callbacks. This is very useful for spec-style tests.

```
<?php
// SomeClass.spec.php

use Demeanor\TestContext;

$this->before(function (TestContext $ctx) {
    $ctx['something'] = createSomeObjectHere();
});

$this->after(function (TestContext $ctx) {
    $ctx['something']->cleanup();
});

$this->it('should do something', function (TestContext $ctx) {
    // use $ctx['something'] here
});
```

Or using *Annotations* with unit tests.

```php
<?php
// AnotherTest.php

use Demeanor\TestContext;

/**
 * @Before(method="setUp")
 * @After(method="tearDown")
 */
class AnotherTest
{
    public function setUp(TestContext $ctx)
    {
        $ctx['something'] = createSomeObjectHere();
    }

    public function tearDown(TestContext $ctx)
    {
        $ctx['something']->tearDown();
    }

    public function testSomething(TestContext $ctx)
    {
        // use $ctx['something']
    }
}
```

1.3 Writing PHPT Tests

`phpt` is a special file layout that the php core uses for its tests.

1.3.1 PHPT Basics

A `.phpt` file is separated into sections by headers that look like `--([A-Z]+)--` where `([A-Z]+)` is replaced with some sequents of one or more uppercase characters.

A bare minimum of the follow sections are required.

- `--TEST--`: Describes the test
- `--FILE--`: The actual php code to run (including the open `<?php` tag)
- `--EXPECT--` or `--EXPECTF--`: Something to match the output of the `--FILE--` section againts.

Here is a valid `.php` file

```
--TEST--
This describe what the file is meant to test: outputing hello world.
--FILE--
<?php
// You need the open php tag here
echo 'Hello, World';
--EXPECT--
Hello, World
```

This is a test that would pass. Why? Because Demeanor will execute the code in the `--FILE--` section in a separate PHP process and compare it with the `--EXPECT--` section. If they match: test passes.

--EXPECTF-- can also be used to match output. This is a details of --EXPECTF--'s format can be found [here](http://qa.php.net/phpt_details.php), but here's quick overview.

- %e: A directory separator (DIRECTORY_SEPARATOR)
- %s: One or more of anything (character, whitespace, etc) except the end of line character. [^rn]+
- %S: Zero or more of anything (character, whatespace, etc) except the end of line character. [^rn]*
- %a: One or more of anything (character, whitespace, etc) including the end of line character. .+
- %A: Zero or more of anything, including the end of line character. .*
- %w: Zero or more whitespace characters. s*
- %i: A signed integer value (+123, -123). [+-]?d+
- %d: An unsigned integer value. d+
- %x: One or more hexadecimal character. [0-9a-fA-F]+
- %f: A floating point number. [+-]?.?d+.?d*(?:[Ee][+-]?d+)?
- %c: A single character. .
- %unicode|string% or %string|unicode%: Matches 'string'
- %binary_string_optional% and %unicode_string_optional%: Matches 'string'
- %u|b% or %b|u%: replaced with nothing

We could rewrite the example above to use --EXPECTF-- so it doesn't care whether it sees "World" or anything else

```
--TEST--
This describe what the file is meant to test: outputing hello world.
--FILE--
<?php
// You need the open php tag here
echo 'Hello, World';
--EXPECT--
Hello, %s
```

1.3.2 Skipping PHPT Tests

Use the --SKIPIF-- section. This is a bit of code that will be pased to a separate PHP process. If the output from it start with *skip* the test will be skipped.

```
--TEST--
Only Runs on php less than 5.4
--SKIPIF--
<?php if (version_compare(phpversion(), '5.4', '<')) {
    echo 'skip on php less than 5.4';
}
--FILE--
<?php
// test code here
--EXPECT--
some sort of output
```

1.3.3 Cleaning Up

Use the `--CLEAN--` section to clean up after yourself. Please note that the `--CLEAN--` section is **not** passed to the same PHP process as the `--FILE--`, so you can expect it to have the same variables available

1.3.4 Sharing Environment

If a file has the optional `--ENV--` section, it will parsed into an associative array and passed to all PHP processes as environment variables.

```
--TEST--
Test with environment
--ENV--
FROM_PHPT_ENV=1
--FILE--
<?php
var_dump(getenv('FROM_PHT_ENV'));
--EXPECTF--
%string|unicode%(%d) "1"
```

The `FROM_PHPT_ENV` will be available (via `getenv` or `$_ENV`, depending on your php settings) in `--SKIPIF--`, `--FILE--`, and `--CLEAN--`.

1.3.5 Does Demeanor Support All PHPT Features?

Definitely not. The PHP core's `run-tests.php` is still much, much more complete. Demeanor just barely does an impression of the phpt functionality found there.

1.4 Annotations

Demeanor uses a simple annotation library to make it a bit easier to configure unit tests.

Annotations are case sensitive.

1.4.1 Annotations on Classes or Methods?

Annotations can be defined on the test class or test method. Annotations on the class will be applied to all test methods in the class. A great use case for this is running a method before every test.

```
<?php
// AnnotationTest.php

/**
 * @Before(method="setUp")
 */
class AnnotationTest
{
    public function setUp()
    {

    }
}
```

1.4.2 Adding Before/After Callbacks

The `Before` and `After` annotations provide ways to call methods on the test class or some function before and after each test case.

```
<?php
// BeforeTest.php

function run_before_example()
{

}

class BeforeTest
{
    public function setUp()
    {

    }

    /**
     * These two are the same:
     * @Before(method="setUp")
     * @Before("setUp")
     *
     * @Before(function="run_before_example")
     */
    public function testStuff()
    {
        // the `setUp` method is run before the `testStuff` method
        // the `run_before_example` function is also run before the
        // `testStuff method
    }
}
```

Both `Before` and `After` can take a `method` OR `function` argument. As you might expect, `method` calls a method on the test class before the test run and `function` calls a function.

Nothing will be added if the `method` doesn't exist or isn't public or if the function doesn't exist.

1.4.3 Expecting Exceptions

The `Expect` annotation can be used instead of calling `TestContext::expectException` in the test method. `Expect` requires the `exception` argument to work.

```
<?php
// ExpectTest.php

use Demeanor\TestContext;

class ExpectTest
{
    /**
     * These two are the same:
     * @Expect("InvalidArgumentException")
     * @Expect(exception="InvalidArgumentException")
     */
    public function testDoingSomethingThrowsException(TestContext $ctx)
```

```
    {
        // same as calling $ctx->expectException('InvalidArgumentException');
    }
}
```

If the class name in the `exception` argument doesn't exist, the test will be errored and will show an error message saying that the exception class wasn't found.

1.4.4 Specifying Requirements

See the *Requirements* documentation for information about using annotations to specify requirements.

1.4.5 Data Providers

Data providers can also be specified with annotations. Details on them can be found on the *data providers* page.

1.5 Data Providers

Data providers let you run the same test with multiple sets of arguments. Any `TestCase` implementation can have a provider, but currently they are only really first class citizens in the unit test world on Demeanor.

1.5.1 Annotations

Data providers are set on unit tests using the `@Provider` annotation.

Data providers can be one of three types:

1. A static method on the test class – `@Provider(method="someMethod")` or `@Provider("someMethod")`
2. A function – `@Provider(function="a_provider_function")`
3. Inline
    - `@Provider(data=["one", "two"])`
    - `@Provider(data={aKey: ["data", "set"], anotherKey: "dataset"})`

1.5.2 Static Method Data Provider

```
<?php
use Demeanor\TestContext;
use Counterpart\Assert;

class DataProviderMethodTest
{
    public static function aProvider()
    {
        return [
            'one',
            'two',
        ];
    }
```

```
    /**
     * These two are the same:
     * @Provider("aProvider")
     * @Provider(method="aProvider")
     */
    public function testWithMethodProvider(TestContext $ctx, $arg)
    {
        Assert::assertType('string', $arg);
    }
}
```

1.5.3 Function Data Provider

```
<?php
use Demeanor\TestContext;
use Counterpart\Assert;

function acceptance_dataprovider_function()
{
    return [
        'one',
        'two',
    ];
}

class DataProviderFunctionTest
{
    /**
     * @Provider(function="acceptance_dataprovider_function")
     */
    public function testWithFunctionProvider(TestContext $ctx, $arg)
    {
        Assert::assertType('string', $arg);
    }
}
```

1.5.4 Inline Data Provider

```
<?php
use Demeanor\TestContext;
use Counterpart\Assert;

class DataProviderInlineTest
{
    /**
     * @Provider(data=["one", "two"])
     */
    public function testWithDataProviderAsIndexedArray(TestContext $ctx, $arg)
    {
        Assert::assertType('string', $arg);
    }

    /**
     * @Provider(data={aSet: "one", anotherSet: "two"})
```

```
     */
    public function testWithDataProviderAsAssociativeArray(TestContext $ctx, $arg)
    {
        Assert::assertType('string', $arg);
    }
}
```

1.5.5 The Test Context

Notice the the *test context* is *always* the first argument to test methods. In Demeanor the context object is important, and any data provider arguments will come after it.

1.6 Requirements

Requirements conditions that have to be met for a test to run. Things like PHP version, a required extension, or a specific OS.

1.6.1 Setting Requirements via the Test Context

The `TestContext` object for a test case will have the key `requirements` that can be used used to add new requirements to a test. This should be done in a before callback.

Unit Test Example

```
<?php
// SomeTest.php

use Demeanor\TestContext;
use Demeanor\Extension\Requirement\VersionRequirement;
use Demeanor\Extension\Requirement\RegexRequirement;
use Demeanor\Extension\Requirement\ExtensionRequirement;

class SomeTest
{
    public function beforeTest(TestContext $ctx)
    {
        // require PHP 5.4
        $ctx['requirements']->add(new VersionRequirement('5.4'));

        // requires a specific verison of some other software
        $ctx['requirements']->add(new VersionRequirement('1.0', getTheVersion(), 'Software Name'));

        // require a specific OS
        $ctx['requirements']->add(new RegexRequirement('/darwin/u', PHP_OS, 'operating system'));

        // require an extension
        $ctx['requirements']->add(new ExtensionRequirement('apc'));
    }

    /**
     * @Before(method="beforeTest")
     */
```

```
    public function testSomeStuff()
    {
        // requirements are checked before this is run
    }
}
```

Unit test requirements can also be set with an annotation. See the *Annotations* documentation for examples.

Spec Test Example

```
<?php
// Some.spec.php

use Demeanor\TestContext;
use Demeanor\Extension\Requirement\VersionRequirement;
use Demeanor\Extension\Requirement\RegexRequirement;
use Demeanor\Extension\Requirement\ExtensionRequirement;

$this->before(function (TestContext $ctx) {
    // require PHP 5.4
    $ctx['requirements']->add(new VersionRequirement('5.4'));

    // requires a specific verison of some other software
    $ctx['requirements']->add(new VersionRequirement('1.0', getTheVersion(), 'Software Name'));

    // require a specific OS
    $ctx['requirements']->add(new RegexRequirement('/darwin/u', PHP_OS, 'operating system'));

    // require an extension
    $ctx['requirements']->add(new ExtensionRequirement('apc'));
});

$this->it('should do something', function (TestContext $ctx) {
    // requirements are checked before this is run
});
```

1.6.2 Setting Requirements via Annotations

Unit test requirements can be set with an annotation. These are limited to PHP version, OS, and extension requirements.

```
<?php
// SomeOtherTest.php

use Demeanor\TestContext;

class SomeOtherTest
{
    /**
     * @Require(php="5.4", os="/darwin/u", extension="apc")
     */
    public function testSomeStuff()
    {
        // requirements are checked before this is run
    }
```

```
    /**
     * Or each Require annotation can be separate
     *
     * @Require(php="5.4")
     * @Require(os="/darwin/u")
     * @Require(extension="apc")
     */
    public function testSomeOtherStuff()
    {
        // requirements are checked before this is run
    }
}
```

1.7 Mock Objects

Use Mockery. Demeanor takes care of verifying mock expectations for you.

1.8 Test Result Statuses

Every `Demeanor\TestCase` implementation produces an implementation of `Demeanor\TestResult` when it's run. `TestResult` can have one of many statuses that are explained here.

1.8.1 Successful

The test worked! To demeanor this just means that no exceptions were thrown during the core of the test run.

1.8.2 Skipped

The test was explicitly marked as skipped via `Demeanor\TestContext::skip` or some sort of requirement for the tests execution was not met.

A skipped test does not cause the Demanor CLI to exit with a failure code.

Users can skip tests if some precondition is not met. Need a certain environment variable for a test to work? Didn't get it? Skip the test.

1.8.3 Errored

An unexpected exception or warning occurred during the execution of the test.

1.8.4 Failed

The test was explicitly marked as failed via `Demeanor\TestContext::fail` or an assertion failed.

1.8.5 Filtered

This status is only used internally by demeanor to "skip" tests with out really skipping them. Filtered means some filter condition (like name or otherwise) was not met and the test was simply not executed.

1.8.6 How Test Result Statuses Influence CLI Exit Codes

If one or more tests fail or error, Demeanor will exit unsuccessfully. See *Exit Codes* for more information.

1.9 Exit Codes

When all tests are successful, Demeanor will exit with the status code `0`.

If tests fail or error, the exit code will `1`.

If some sort of error happens (configuration issue, etc) before tests are run, the exit code will be `2`.

If, for some reason, the `demeanor` command line script can't find a composer autoload file, it will exit with the status code `3`.

CHAPTER 2

Getting Started

Demeanor can be installed with composer, please read the composer getting started page to learn how to get everything set up.

Once that's done, add `demeanor/demeanor` to your `require-dev` dependencies in `composer.json`.

```
{
    "require-dev": {
        "demeanor/demeanor": "dev-master"
    }
}
```

Then run `composer install` or `composer update` with the `--dev` flag.

2.1 Configuration

Other documentation explains the `demeanor.json` configuration more fully, but, for now, we're going to set up two test suites.

```
{
    "testsuites": {
        "unit": {
            "type": "unit",
            "directories": [
                "test/unit"
            ]
        },
        "spec": {
            "type": "spec",
            "directories": [
                "test/spec"
            ]
        }
    }
}
```

The `testsuites` argument is required and **must** be an object. If it's not the test runner will complain.

The keys of the `testsuites` object are the suite names and the values are their configuration. `type` tells Demeanor what type of test suite its dealing with. Valid values are `unit`, `spec`, or `phpt`. `directories` tells the test run where to look for the test files. How Demeanor finds those files varies by suite type.

2.2 Assertions

Demeanor uses a library called Counterpart to deal with assertions. You'll use the `Counterpart\Assert` class and call one of it's `assert*` methods. The last argument of all `assert*` methods is a message that can be used to describe the business case or importances of the assertion.

Here are some examples:

```
<?php
use Counterpart\Assert;

Assert::assertTrue(true, 'True is somehow false, things are very broken');
Assert::assertFalse(false);
Assert::assertNull(null);
Assert::assertType('string', 'this is a string');
```

2.3 Your First Unit Test

Unit test cases are methods inside of a class. Every time a method is run, a new instance of it's class is created.

Test class names **must** end with `Test` and test method must start with the word `test`. Demeanor will look for all files that end with `Test.php` in the directories defined in the `directories` configuration above.

```
<?php
// test/unit/TruthyTest.php

use Counterpart\Assert;

class TruthyTest
{
    public function testTruthyValuesReturnTrue()
    {
        Assert::assertTrue(filter_var('yes', FILTER_VALIDATE_BOOLEAN));
    }

    public function testFalsyValuesReturnFalse()
    {
        Assert::assertFalse(filter_var('no', FILTER_VALIDATE_BOOLEAN));
    }
}
```

2.4 Your First Spec Test

Spec tests use a `describe` and `it` API to to define a specification for an object. A specification is just a set of expected behaviors.

In demeanor, a spec test looks like this.

```
<?php
// filter_var.spec.php

use Counterpart\Assert;

/** @var Demeanor\Spec\Specification $this */
```

```
$this->describe('#truthyValues', function () {
    $this->it('should return true when given "yes"', function () {
        Assert::assertTrue(filter_var('yes', FILTER_VALIDATE_BOOLEAN));
    });
    $this->it('should return true when given a "1"', function () {
        Assert::assertTrue(filter_var(1, FILTER_VALIDATE_BOOLEAN));
    });
});

$this->describe('#falsyValues', function () {
    $this->it('should return false when given "no"', function () {
        Assert::assertFalse(filter_var('no', FILTER_VALIDATE_BOOLEAN));
    });
    $this->it('should return false when given "0"', function () {
        Assert::assertFalse(filter_var(0, FILTER_VALIDATE_BOOLEAN));
    });
});
```

Each call to `it` creates a new test case. When the `directories` argument is used for a `spec` test suite, all files that end with `.spec.php` are located and compiled to test cases.

2.5 Running the Tests

A binary will be installed via composer in your `bin-dir` (`vendor/bin` by default). Once a configuration file and some tests are set up, use the command line to run `php vendor/bin/demeanor` or `./vendor/bin/demeanor` to run the tests.

2.6 Need More Examples?

Demeanor uses itself to test – well, to test itself. Look in the `test` directory of the demeanor repository for a bunch more examples.