Unit Tests

Codeception uses PHPUnit as a backend for running its tests. Thus, any PHPUnit test can be added to a Codeception test suite and then executed. If you ever wrote a PHPUnit test then do it just as you did before. Codeception adds some nice helpers to simplify common tasks.

The basics of unit tests are skipped here, instead you will get a basic knowledge of which features Codeception adds to unit tests.

To say it again: you don't need to install PHPUnit to run its tests. Codeception can run them too.

Creating Test

Create a test using `generate:test` command with a suite and test names as parameters:

```
php codecept generate:test unit Example
```

It creates a new `ExampleTest` file located in the `tests/unit` directory.

As always, you can run the newly created test with this command:

```
php codecept run unit ExampleTest
```

Or simply run the whole set of unit tests with:

```
php codecept run unit
```

A test created by the `generate:test` command will look like this:

```
<?php

class ExampleTest extends \Codeception\Test\Unit
{
    /**
     * @var \UnitTester
     */
    protected $tester;

    protected function _before()
    {
    }

    protected function _after()
    {
    }

    // tests
    public function testMe()
    {

    }
}
```

This class has predefined `_before` and `_after` methods. You can use them to create a tested object before each test, and destroy it afterwards.

As you see, unlike in PHPUnit, the `setUp` and `tearDown` methods are replaced with their aliases: `_before`, `_after`.

The actual `setUp` and `tearDown` are implemented by the parent class `\Codeception\TestCase\Test` and set the UnitTester class up to have all the cool actions from Cept-files to be run as a part of your unit tests. Just like in the acceptance and functional tests, you can choose the proper modules for the `UnitTester` class in the `unit.suite.yml` configuration file:

```
# Codeception Test Suite Configuration

# suite for unit (internal) tests.
actor: UnitTester
modules:
    enabled:
        - Asserts
        - \Helper\Unit
```

Classical Unit Testing

Unit tests in Codeception are written in absolutely the same way as in PHPUnit:

```
<?php
class UserTest extends \Codeception\Test\Unit
{
    public function testValidation()
    {
        $user = User::create();

        $user->username = null;
        $this->assertFalse($user->validate(['username']));

        $user->username = 'tooloooooongnaaaaaaameeee';
        $this->assertFalse($user->validate(['username']));

        $user->username = 'davert';
        $this->assertTrue($user->validate(['username']));
    }
}
```

Using Modules

As in scenario-driven functional or acceptance tests you can access Actor class methods. If you write integration tests, it may be useful to include the `Db` module for database testing.

```
# Codeception Test Suite Configuration

# suite for unit (internal) tests.
actor: UnitTester
modules:
    enabled:
        - Asserts
        - Db
        - \Helper\Unit
```

To access UnitTester methods you can use the `UnitTester` property in a test.

Testing Database

Let's see how you can do some database testing:

```
<?php
function testSavingUser()
{
    $user = new User();
    $user->setName('Miles');
    $user->setSurname('Davis');
    $user->save();
    $this->assertEquals('Miles Davis', $user->getFullName());
    $this->tester->seeInDatabase('users', ['name' => 'Miles', 'surname' => 'Davis']);
}
```

To enable the database functionality in unit tests, make sure the `Db` module is included in the `unit.suite.yml` configuration file. The database will be cleaned and populated after each test, the same way it happens for acceptance and functional tests. If that's not your required behavior, change the settings of the `Db` module for the current suite. See Db Module (http://codeception.com/docs/modules/Db)

Interacting with the Framework

You should probably not access your database directly if your project already uses ORM for database interactions. Why not use ORM directly inside your tests? Let's try to write a test using Laravel's ORM Eloquent. For this we need to configure the Laravel5 module. We won't need its web interaction methods like `amOnPage` or `see`, so let's enable only the ORM part of it:

```
actor: UnitTester
modules:
    enabled:
        - Asserts
        - Laravel5:
            part: ORM
        - \Helper\Unit
```

We included the Laravel5 module the same way we did for functional testing. Let's see how we can use it for integration tests:

```
<?php
function testUserNameCanBeChanged()
{
    // create a user from framework, user will be deleted after the test
    $id = $this->tester->haveRecord('users', ['name' => 'miles']);
    // access model
    $user = User::find($id);
    $user->setName('bill');
    $user->save();
    $this->assertEquals('bill', $user->getName());
    // verify data was saved using framework methods
    $this->tester->seeRecord('users', ['name' => 'bill']);
    $this->tester->dontSeeRecord('users', ['name' => 'miles']);
}
```

A very similar approach can be used for all frameworks that have an ORM implementing the ActiveRecord pattern. In Yii2 and Phalcon, the methods `haveRecord`, `seeRecord`, `dontSeeRecord` work in the same way. They also should be included by specifying `part: ORM` in order to not use the functional testing actions.

If you are using Symfony with Doctrine, you don't need to enable Symfony itself but just Doctrine2:

```
actor: UnitTester
modules:
    enabled:
        - Asserts
        - Doctrine2:
            depends: Symfony
        - \Helper\Unit
```

In this case you can use the methods from the Doctrine2 module, while Doctrine itself uses the Symfony module to establish connections to the database. In this case a test might look like:

```
<?php
function testUserNameCanBeChanged()
{
    // create a user from framework, user will be deleted after the test
    $id = $this->tester->haveInRepository('Acme\DemoBundle\Entity\User', ['name' => 'miles']);
    // get entity manager by accessing module
    $em = $this->getModule('Doctrine2')->em;
    // get real user
    $user = $em->find('Acme\DemoBundle\Entity\User', $id);
    $user->setName('bill');
    $em->persist($user);
    $em->flush();
    $this->assertEquals('bill', $user->getName());
    // verify data was saved using framework methods
    $this->tester->seeInRepository('Acme\DemoBundle\Entity\User', ['name' => 'bill']);
    $this->tester->dontSeeInRepository('Acme\DemoBundle\Entity\User', ['name' => 'miles']);
}
```

In both examples you should not be worried about the data persistence between tests. The Doctrine2 and Laravel5 modules will clean up the created data at the end of a test. This is done by wrapping each test in a transaction and rolling it back afterwards.

Accessing Module

Codeception allows you to access the properties and methods of all modules defined for this suite. Unlike using the UnitTester class for this purpose, using a module directly grants you access to all public properties of that module.

We have already demonstrated this in a previous example where we accessed the Entity Manager from a Doctrine2 module:

```
<?php
/** @var Doctrine\ORM\EntityManager */
$em = $this->getModule('Doctrine2')->em;
```

If you use the `Symfony` module, here is how you can access the Symfony container:

```
<?php
/** @var Symfony\Component\DependencyInjection\Container */
$container = $this->getModule('Symfony')->container;
```

The same can be done for all public properties of an enabled module. Accessible properties are listed in the module reference.

BDD Specification Testing

When writing tests you should prepare them for constant changes in your application. Tests should be easy to read and maintain. If a specification of your application is changed, your tests should be updated as well. If you don't have a convention inside your team for documenting tests, you will have issues figuring out what tests will be affected by the introduction of a new feature.

That's why it's pretty important not just to cover your application with unit tests, but make unit tests self-explanatory. We do this for scenario-driven acceptance and functional tests, and we should do this for unit and integration tests as well.

For this case we have a stand-alone project Specify (https://github.com/Codeception/Specify) (which is included in the phar package) for writing specifications inside unit tests:

```
<?php
class UserTest extends \Codeception\Test\Unit
{
    use \Codeception\Specify;

    private $user;

    public function testValidation()
    {
        $this->user = User::create();

        $this->specify("username is required", function() {
            $this->user->username = null;
            $this->assertFalse($this->user->validate(['username']));
        });

        $this->specify("username is too long", function() {
            $this->user->username = 'tooloooooongnaaaaaaameeee';
            $this->assertFalse($this->user->validate(['username']));
        });

        $this->specify("username is ok", function() {
            $this->user->username = 'davert';
            $this->assertTrue($this->user->validate(['username']));
        });
    }
}
```

By using `specify` codeblocks, you can describe any piece of a test. This makes tests much cleaner and comprehensible for everyone in your team.

Code inside `specify` blocks is isolated. In the example above, any changes to `$this->user` (as with any other object property), will not be reflected in other code blocks. Specify uses deep and shallow cloning strategies to save objects between isolated scopes. The downsides of this approach is increased memory consumption (on deep cloning) or incomplete isolation when shallow cloning is used. Please make sure you understand how Specify (https://github.com/Codeception/Specify) works and how to configure it before using it in your tests.

Also, you may add Codeception\Verify (https://github.com/Codeception/Verify) for BDD-style assertions. This tiny library adds more readable assertions, which is quite nice, if you are always confused about which argument in `assert` calls is expected and which one is actual:

```
<?php
verify($user->getName())->equals('john');
```

Stubs

Codeception provides a tiny wrapper over the PHPUnit mocking framework to create stubs easily. Include `\Codeception\Util\Stub` to start creating dummy objects.

In this example we instantiate an object without calling a constructor and replace the `getName` method to return the value *john*:

```
<?php
$user = Stub::make('User', ['getName' => 'john']);
$name = $user->getName(); // 'john'
```

Stubs are created with PHPUnit's mocking framework. Alternatively, you can use Mockery (https://github.com/padraic/mockery) (with Mockery module (https://github.com/Codeception/MockeryModule)), AspectMock (https://github.com/Codeception/AspectMock) or others.

Here's the full reference on the Stub utility class (/docs/reference/Stub).

Conclusion

PHPUnit tests are first-class citizens in test suites. Whenever you need to write and execute unit tests, you don't need to install PHPUnit seperately, but use Codeception directly to execute them. Some nice features can be added to common unit tests by integrating Codeception modules. For most unit and integration testing, PHPUnit tests are enough. They run fast, and are easy to maintain.

- **Next Chapter: ModulesAndHelpers > (/docs/06-ModulesAndHelpers)**
- **Previous Chapter: < FunctionalTests (/docs/04-FunctionalTests)**

Looking for commercial support? Request official consulting service (http://sdclabs.com/codeception?utm_source=codeception.com&utm_medium=docs_bottom&utm_term=link&utm_campaign=reference)