Getting Started

Let's take a look at Codeception's architecture. We'll assume that you have already installed (http://codeception.com/install) it and bootstrapped your first test suites. Codeception has generated three of them: unit, functional, and acceptance. They are well described in the previous chapter (http://codeception.com/docs/01-Introduction). Inside your **/tests** folder you will have three `.yml` config files and three directories with names corresponding to these suites: `unit`, `functional`, `acceptance`. Suites are independent groups of tests with a common purpose.

The Codeception Syntax

Codeception follows simple naming rules to make it easy to remember (as well as easy to understand) its method names.

- **Actions** start with a plain english verb, like "click" or "fill". Examples:

```
<?php
$I->click('Login');
$I->fillField('#input-username', 'John Dough');
$i->pressKey('#input-remarks', 'foo');
```

- **Assertions** always start with "see" or "dontSee". Examples:

```
<?php
$I->see('Welcome');
$I->seeInTitle('My Company');
$i->seeElement('nav');
$i->dontSeeElement('#error-message');
$i->dontSeeInPageSource('<section class="foo">');
```

- **Grabbers** just *read* something from the page, but don't process it. The return value of those are meant to be saved as variables and used later. Example:

```
<?php
$method = $I->grabAttributeFrom('#login-form', 'method');
$I->assertEquals('post', $method);
```

Actors

One of the main concepts of Codeception is representation of tests as actions of a person. We have a UnitTester, who executes functions and tests the code. We also have a FunctionalTester, a qualified tester, who tests the application as a whole, with knowledge of its internals. Lastly we have an AcceptanceTester, a user who works with our application through an interface that we provide.

Methods of actor classes are generally taken from Codeception Modules (http://codeception.com/docs/06-ModulesAndHelpers). Each module provides predefined actions for different testing purposes, and they can be combined to fit the testing environment. Codeception tries to solve 90% of possible testing issues in its modules, so you don't have to reinvent the wheel. We think that you can spend more time on writing tests and less on writing support code to make those tests run. By default, AcceptanceTester relies on PhpBrowser module, which is set in the `tests/acceptance.suite.yml` configuration file:

```
actor: AcceptanceTester
modules:
    enabled:
        - PhpBrowser:
            url: http://localhost/myapp/
        - \Helper\Acceptance
```

In this configuration file you can enable/disable and reconfigure modules for your needs. When you change the configuration, the actor classes are rebuilt automatically. If the actor classes are not created or updated as you expect, try to generate them manually with the `build` command:

```
php codecept build
```

Writing a Sample Scenario

By default tests are written as narrative scenarios. To make a PHP file a valid scenario, its name should have a `Cept` suffix.

Let's say we have created a file `tests/acceptance/SigninCept.php`

We can do that by running the following command:

```
php codecept generate:cept acceptance Signin
```

A scenario always starts with actor class initialization. After that, writing a scenario is just like typing `$I->` and choosing a proper action from the auto-completion list. Let's log in to our website:

```
<?php
$I = new AcceptanceTester($scenario); // actor class initialization
$I->wantTo('login to website');
```

The `wantTo` section describes your scenario in brief. There are additional comment methods that are useful to describe the context of a scenario:

```
<?php
$I = new AcceptanceTester($scenario);
$I->am('user'); // actor's role
$I->wantTo('login to website'); // feature to test
$I->lookForwardTo('access website features for logged-in users'); // result to achieve
```

After we have described the story background, let's start writing a scenario.

We'll assume that we have a 'login' page where we get authenticated by providing a username and password. Then we are sent to a user page, where we see the text `Hello, %username%`. Let's look at how this scenario is written in Codeception:

```
<?php
$I = new AcceptanceTester($scenario);
$I->am('user');
$I->wantTo('login to website');
$I->lookForwardTo('access website features for logged-in users');
$I->amOnPage('/login');
$I->fillField('Username','davert');
$I->fillField('Password','qwerty');
$I->click('Login');
$I->see('Hello, davert');
```

This scenario can probably be read by non-technical people. If you just remove all special chars like braces, arrows and `$`, this test transforms into plain English text:

```
I am user
I wantTo login to website
I lookForwardTo access website features for logged-in users
I amOnPage '/login'
I fillField 'Username','davert'
I fillField 'Password','qwerty'
I click 'Login'
I see 'Hello, davert'
```

Codeception generates this text representation from PHP code by executing:

```
php codecept generate:scenarios
```

These generated scenarios will be stored in your `_data` directory in text files.

Before we execute this test, we should make sure that the website is running on a local web server. Let's open the `tests/acceptance.suite.yml` file and replace the URL with the URL of your web application:

```
actor: AcceptanceTester
modules:
    enabled:
        - PhpBrowser:
            url: 'http://myappurl.local'
        - \Helper\Acceptance
```

After configuring the URL we can run this test with the `run` command:

```
php codecept run
```

This is the output we should see:

```
Acceptance Tests (1) ------------------------------
✓ SigninCept: Login to website
----------------------------------------------------

Time: 1 second, Memory: 21.00Mb

OK (1 test, 1 assertions)
```

Let's get some detailed output:

```
php codecept run acceptance --steps
```

We should see a step-by-step report on the performed actions:

```
Acceptance Tests (1) ------------------------------
SigninCept: Login to website
Signature: SigninCept.php
Test: tests/acceptance/SigninCept.php
Scenario --
 I am user
 I look forward to access website features for logged-in users
 I am on page "/login"
 I fill field "Username" "davert"
 I fill field "Password" "qwerty"
 I click "Login"
 I see "Hello, davert"
 OK
----------------------------------------------------

Time: 0 seconds, Memory: 21.00Mb

OK (1 test, 1 assertions)
```

This simple test can be extended to a complete scenario of site usage, therefore, by emulating the user's actions, you can test any of your websites.

Give it a try!

Cept, Cest and Test Formats

Codeception supports three test formats. Beside the previously described scenario-based Cept format, Codeception can also execute PHPUnit test files for unit testing (http://codeception.com/docs/05-UnitTests), and Cest format.

Cest combines scenario-driven test approach with OOP design. In case you want to group a few testing scenarios into one, you should consider using Cest format. In the example below we are testing CRUD actions within a single file but with several tests (one per operation):

```
<?php
class PageCrudCest
{
    function _before(AcceptanceTester $I)
    {
        // will be executed at the beginning of each test
        $I->amOnPage('/');
    }

    function createPage(AcceptanceTester $I)
    {
       // todo: write test
    }

    function viewPage(AcceptanceTester $I)
    {
       // todo: write test
    }

    function updatePage(AcceptanceTester $I)
    {
        // todo: write test
    }

    function deletePage(AcceptanceTester $I)
    {
       // todo: write test
    }
}
```

Cest files such as this can be created by running a generator:

```
php codecept generate:cest acceptance PageCrud
```

Learn more about the Cest format (http://codeception.com/docs/07-AdvancedUsage#Cest-Classes) in the Advanced Testing section.

BDD

Codeception allows execution of user stories in Gherkin format in a similar manner as is done in Cucumber or Behat. Please refer to the BDD chapter (http://codeception.com/docs/07-BDD) to learn more.

Configuration

Codeception has a global configuration in `codeception.yml` and a config for each suite. We also support `.dist` configuration files. If you have several developers in a project, put shared settings into `codeception.dist.yml` and personal settings into `codeception.yml`. The same goes for suite configs. For example, the `unit.suite.yml` will be merged with `unit.suite.dist.yml`.

Running Tests

Tests can be started with the `run` command:

```
php codecept run
```

With the first argument you can run all tests from one suite:

```
php codecept run acceptance
```

To limit tests run to a single class, add a second argument. Provide a local path to the test class, from the suite directory:

```
php codecept run acceptance SigninCept.php
```

Alternatively you can provide the full path to test file:

```
php codecept run tests/acceptance/SigninCept.php
```

You can further filter which tests are run by appending a method name to the class, separated by a colon (for Cest or Test formats):

```
php codecept run tests/acceptance/SignInCest.php:^anonymousLogin$
```

You can provide a directory path as well. This will execute all acceptance tests from the `backend` dir:

```
php codecept run tests/acceptance/backend
```

Using regular expressions, you can even run many different test methods from the same directory or class. For example, this will execute all acceptance tests from the `backend` dir beginning with the word “login”:

```
php codecept run tests/acceptance/backend:^login
```

To execute a group of tests that are not stored in the same directory, you can organize them in groups (http://codeception.com/docs/07-AdvancedUsage#Groups).

Reports

To generate JUnit XML output, you can provide the `--xml` option, and `--html` for HTML report.

```
php codecept run --steps --xml --html
```

This command will run all tests for all suites, displaying the steps, and building HTML and XML reports. Reports will be stored in the `tests/_output/` directory.

To see all the available options, run the following command:

```
php codecept help run
```

Debugging

To receive detailed output, tests can be executed with the `--debug` option. You may print any information inside a test using the `codecept_debug` function.

Generators

There are plenty of useful Codeception commands:

- `generate:cept` *suite filename* - Generates a sample Cept scenario
- `generate:cest` *suite filename* - Generates a sample Cest test

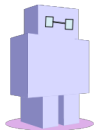
- `generate:test` *suite filename* - Generates a sample PHPUnit Test with Codeception hooks
- `generate:feature` *suite filename* - Generates Gherkin feature file
- `generate:suite` *suite actor* - Generates a new suite with the given Actor class name
- `generate:scenarios` *suite* - Generates text files containing scenarios from tests
- `generate:helper` *filename* - Generates a sample Helper File
- `generate:pageobject` *suite filename* - Generates a sample Page object
- `generate:stepobject` *suite filename* - Generates a sample Step object
- `generate:environment` *env* - Generates a sample Environment configuration
- `generate:groupobject` *group* - Generates a sample Group Extension

Conclusion

We have taken a look into the Codeception structure. Most of the things you need were already generated by the `bootstrap` command. After you have reviewed the basic concepts and configurations, you can start writing your first scenario.

- **Next Chapter: AcceptanceTests > (/docs/03-AcceptanceTests)**
- **Previous Chapter: < Introduction (/docs/01-Introduction)**

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)



(http://robo.li)

Modern PHP **Task Runner**. Allows to declare tasks with zero configuration in pure PHP.

(http://robo.li)

CodeceptJS

(http://codecept.io)

(http://codecept.io) Codeception for **NodeJS**. Write acceptance tests in ES6 and execute in webdriverio, Selenium WebDriver, and Protractor.