CODECEPTION\_

GUIDES

opencollective

2020

Índice

[GETTING STARTED 7](#_Toc60559899)

[THE CODECEPTION SYNTAX 7](#_Toc60559900)

[ACTORS 8](#_Toc60559901)

[WRITING A SAMPLE TEST 8](#_Toc60559902)

[INTERACTIVE PAUSE 13](#_Toc60559903)

[BDD 15](#_Toc60559904)

[CONFIGURATION 15](#_Toc60559905)

[RUNNING TESTS 15](#_Toc60559906)

[REPORTS 16](#_Toc60559907)

[DEBUGGING 16](#_Toc60559908)

[GENERATORS 16](#_Toc60559909)

[CONCLUSION 17](#_Toc60559910)

[YII BASIC 18](#_Toc60559911)

[UNIT TESTS 18](#_Toc60559912)

[FUNCTIONAL TESTS 20](#_Toc60559913)

[API TESTS 21](#_Toc60559914)

[ACCEPTANCE TESTS 22](#_Toc60559915)

[YII ADVANCED 23](#_Toc60559916)

[MANUAL SETUP && CONFIGURATION 24](#_Toc60559917)

[ACCEPTANCE TESTING 26](#_Toc60559918)

[SAMPLE SCENARIO 26](#_Toc60559919)

[PHPBROWSER 27](#_Toc60559920)

[WEBDRIVER 34](#_Toc60559921)

[LOCAL SETUP 35](#_Toc60559922)

[CONFIGURATION 36](#_Toc60559923)

[MULTI SESSION TESTING 41](#_Toc60559924)

[CLOUD TESTING 42](#_Toc60559925)

[DEBUGGING 43](#_Toc60559926)

[COMMON CASES 43](#_Toc60559927)

[CUSTOM BROWSER SESSIONS 45](#_Toc60559928)

[CONCLUSION 46](#_Toc60559929)

[FUNCTIONAL TESTS 47](#_Toc60559930)

[LIMITATIONS 48](#_Toc60559931)

[HEADERS, COOKIES, SESSIONS 48](#_Toc60559932)

[EXTERNAL URLS 48](#_Toc60559933)

[SHARED MEMORY 48](#_Toc60559934)

[ENABLING FRAMEWORK MODULES 48](#_Toc60559935)

[SYMFONY 48](#_Toc60559936)

[LARAVEL5 49](#_Toc60559937)

[YII2 49](#_Toc60559938)

[ZEND FRAMEWORK 2 49](#_Toc60559939)

[ZEND EXPRESSIVE 50](#_Toc60559940)

[PHALCON 4 50](#_Toc60559941)

[WRITING FUNCTIONAL TESTS 51](#_Toc60559942)

[ERROR REPORTING 52](#_Toc60559943)

[CONCLUSION 53](#_Toc60559944)

[UNIT & INTEGRATION TESTS 54](#_Toc60559945)

[CREATING A TEST 54](#_Toc60559946)

[UNIT TESTING 55](#_Toc60559947)

[ASSERTIONS 56](#_Toc60559948)

[TEST DOUBLES 56](#_Toc60559949)

[INTEGRATION TESTS 59](#_Toc60559950)

[USING MODULES 59](#_Toc60559951)

[TESTING DATABASE 59](#_Toc60559952)

[INTERACTING WITH THE FRAMEWORK 60](#_Toc60559953)

[ACCESSING MODULE 62](#_Toc60559954)

[SCENARIO DRIVEN TESTING 63](#_Toc60559955)

[ADVANCED TOOLS 64](#_Toc60559956)

[SPECIFY 64](#_Toc60559957)

[DOMAIN ASSERTIONS 65](#_Toc60559958)

[ASPECTMOCK 66](#_Toc60559959)

[CONCLUSION 67](#_Toc60559960)

[MODULES AND HELPERS 68](#_Toc60559961)

[STANDARD MODULES 69](#_Toc60559962)

[MODULE CONFLICTS 69](#_Toc60559963)

[MODULE PARTS 70](#_Toc60559964)

[HELPERS 71](#_Toc60559965)

[ACCESSING OTHER MODULES 73](#_Toc60559966)

[EXTENDING A MODULE 74](#_Toc60559967)

[HOOKS 75](#_Toc60559968)

[DEBUG 76](#_Toc60559969)

[CONFIGURATION 77](#_Toc60559970)

[DYNAMIC CONFIGURATION WITH PARAMETERS 78](#_Toc60559971)

[RUNTIME CONFIGURATION 80](#_Toc60559972)

[RUNTIME CONFIGURATION OF A TEST 80](#_Toc60559973)

[CONCLUSION 81](#_Toc60559974)

[REUSING TEST CODE 82](#_Toc60559975)

[WHAT ARE ACTORS 82](#_Toc60559976)

[**PAGEOBJECTS** 84](#_Toc60559977)

[**STEPOBJECTS** 86](#_Toc60559978)

[**CONCLUSION** 89](#_Toc60559979)

[ADVANCED USAGE 90](#_Toc60559980)

[CEST CLASSES 90](#_Toc60559981)

[DEPENDENCY INJECTION 92](#_Toc60559982)

[EXAMPLE ANNOTATION 95](#_Toc60559983)

[DATAPROVIDER ANNOTATIONS 97](#_Toc60559984)

[BEFORE/AFTER ANNOTATIONS 98](#_Toc60559985)

[ENVIRONMENTS 100](#_Toc60559986)

[GET SCENARIO METADATA 102](#_Toc60559987)

[SHUFFLE 104](#_Toc60559988)

[DEPENDENCIES 104](#_Toc60559989)

[RUNNING FROM DIFFERENT FOLDERS 105](#_Toc60559990)

[GROUPS 106](#_Toc60559991)

[GROUP FILES 107](#_Toc60559992)

[FORMATS 108](#_Toc60559993)

[SHELL AUTO-COMPLETION 109](#_Toc60559994)

[EXPLANATION 109](#_Toc60559995)

[CONCLUSION 110](#_Toc60559996)

[BEHAVIOR DRIVEN DEVELOPMENT 111](#_Toc60559997)

[WHAT IS BEHAVIOR DRIVEN DEVELOPMENT 111](#_Toc60559998)

[UBIQUITOUS LANGUAGE 112](#_Toc60559999)

[GHERKIN 113](#_Toc60560000)

[FEATURES 113](#_Toc60560001)

[STEP DEFINITIONS 116](#_Toc60560002)

[TESTING BEHAVIOR 117](#_Toc60560003)

[ACCEPTANCE TESTING 117](#_Toc60560004)

[ADVANCED GHERKIN 121](#_Toc60560005)

[BACKGROUND 122](#_Toc60560006)

[TABLES 122](#_Toc60560007)

[EXAMPLES 123](#_Toc60560008)

[LONG STRINGS 124](#_Toc60560009)

[TAGS 125](#_Toc60560010)

[CONFIGURATION 125](#_Toc60560011)

[MIGRATING FROM BEHAT 127](#_Toc60560012)

[TESTS VS FEATURES 127](#_Toc60560013)

[CONCLUSIONS 128](#_Toc60560014)

[CUSTOMIZATION 129](#_Toc60560015)

[NAMESPACES 129](#_Toc60560016)

[BOOTSTRAP 129](#_Toc60560017)

[GLOBAL BOOTSTRAP 130](#_Toc60560018)

[SUITE BOOTSTRAP 130](#_Toc60560019)

[ON FLY BOOTSTRAP 130](#_Toc60560020)

[EXTENSION 130](#_Toc60560021)

[EVENTS 131](#_Toc60560022)

[ENABLING EXTENSION 134](#_Toc60560023)

[CONFIGURING EXTENSION 134](#_Toc60560024)

[CUSTOM COMMANDS 135](#_Toc60560025)

[GROUP OBJECTS 135](#_Toc60560026)

[STEP DECORATORS 138](#_Toc60560027)

[CUSTOM REPORTERS 138](#_Toc60560028)

[INSTALLATION TEMPLATES 139](#_Toc60560029)

[ONE RUNNER FOR MULTIPLE APPLICATIONS 140](#_Toc60560030)

[CONCLUSION 141](#_Toc60560031)

[WORKING WITH DATA 142](#_Toc60560032)

[DB 142](#_Toc60560033)

[SEQUENCE 144](#_Toc60560034)

[ORM MODULES 145](#_Toc60560035)

[ACTIVERECORD 145](#_Toc60560036)

[DOCTRINE 147](#_Toc60560037)

[DATAFACTORY 147](#_Toc60560038)

[TESTING DYNAMIC DATA WITH SNAPSHOTS 148](#_Toc60560039)

[FAILED ASSERTION OUTPUT 150](#_Toc60560040)

[WORKING WITH DIFFERENT DATA FORMATS 151](#_Toc60560041)

[CONCLUSION 151](#_Toc60560042)

[API TESTING 152](#_Toc60560043)

[REST API 152](#_Toc60560044)

[AUTHORIZATION 154](#_Toc60560045)

[SENDING REQUESTS 155](#_Toc60560046)

[JSON STRUCTURE VALIDATION 155](#_Toc60560047)

[TAKING DATA FROM RESPONSES 156](#_Toc60560048)

[VALIDATING DATA JSON RESPONSES 157](#_Toc60560049)

[TESTING XML RESPONSES 158](#_Toc60560050)

[SOAP API 159](#_Toc60560051)

[CONCLUSION 161](#_Toc60560052)

[CODE COVERAGE 162](#_Toc60560053)

[CONFIGURATION 163](#_Toc60560054)

[LOCAL CODECOVERAGE 164](#_Toc60560055)

[REMOTE CODECOVERAGE 164](#_Toc60560056)

[LOCAL SERVER 164](#_Toc60560057)

[REMOTE SERVER 165](#_Toc60560058)

[WORKING DIRECTORY (DOCKER/SHARED MOUNTS) 166](#_Toc60560059)

[REMOTE CONTEXT OPTIONS 166](#_Toc60560060)

[CONCLUSION 167](#_Toc60560061)

[CONTINUOUS INTEGRATION 168](#_Toc60560062)

[JENKINS 168](#_Toc60560063)

[PREPARING JENKINS 169](#_Toc60560064)

[BASIC SETUP 170](#_Toc60560065)

[XML REPORTS 171](#_Toc60560066)

[HTML REPORTS 172](#_Toc60560067)

[TEAMCITY 173](#_Toc60560068)

[TRAVISCI 175](#_Toc60560069)

[GITLAB 175](#_Toc60560070)

[CONCLUSION 178](#_Toc60560071)

[PARALLEL EXECUTION 179](#_Toc60560072)

[WHERE TO START 179](#_Toc60560073)

[DOCKER 179](#_Toc60560074)

[USING CODECEPTION DOCKER IMAGE 179](#_Toc60560075)

[ROBO 182](#_Toc60560076)

[WHAT TO DO 182](#_Toc60560077)

[PREPARING ROBO AND ROBO-PARACEPT 182](#_Toc60560078)

[PREPARING ROBO 182](#_Toc60560079)

[CONCLUSION 190](#_Toc60560080)

# GETTING STARTED

Let’s take a look at Codeception’s architecture. We’ll assume that you have already [installed](https://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](https://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:

**$I->click('Login');**

**$I->fillField('#input-username', 'John Dough');**

**$I->pressKey('#input-remarks', 'foo');**

* **Assertions** always start with “see” or “dontSee”. Examples:

**$I->see('Welcome');**

**$I->seeInTitle('My Company');**

**$I->seeElement('nav');**

**$I->dontSeeElement('#error-message');**

**$I->dontSeeInPageSource('<section class="foo">');**

* **Grabbers** take information. The return value of those are meant to be saved as variables and used later. Example:

**$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](https://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 vendor/bin/codecept build**

## WRITING A SAMPLE TEST

Codeception has its own testing format called Cest (Codecept + Test). To start writing a test we need to create a new Cest file. We can do that by running the following command:

**php vendor/bin/codecept generate:cest acceptance Signin**

This will generate SigninCest.php file inside tests/acceptance directory. Let’s open it:

**class SigninCest**

**{**

**function \_before(AcceptanceTester $I)**

**{**

**}**

**public function \_after(AcceptanceTester $I)**

**{**

**}**

**public function tryToTest(AcceptanceTester $I)**

**{**

***// todo: write test***

**}**

**}**

We have \_before and \_after methods to run some common actions before and after a test. And we have a placeholder action tryToTest which we need to implement. If we try to test a signin process it’s a good start to test a successful signin. Let’s rename this method to signInSuccessfully.

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:

**class SigninCest**

**{**

**public function signInSuccessfully(AcceptanceTester $I)**

**{**

**$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 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 vendor/bin/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 vendor/bin/codecept run**

This is the output we should see:

**Acceptance Tests (1) -------------------------------**

**✔ SigninCest: sign in successfully**

**----------------------------------------------------**

**Time: 1 second, Memory: 21.00Mb**

**OK (1 test, 1 assertions)**

Let’s get some detailed output:

**php vendor/bin/codecept run acceptance --steps**

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

**Acceptance Tests (1) -------------------------------**

**SigninCest: Login to website**

**Signature: SigninCest.php:signInSuccessfully**

**Test: tests/acceptance/SigninCest.php:signInSuccessfully**

**Scenario --**

**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.

To run more tests create a public method for each of them. Include AcceptanceTester object as $I as a method parameter and use the same $I-> API you’ve seen before. If your tests share common setup actions put them into \_before method.

For instance, to test CRUD we want 4 methods to be implemented and all next tests should start at /task page:

**class TaskCrudCest**

**{**

**function \_before(AcceptanceTester $I)**

**{**

***// will be executed at the beginning of each test***

**$I->amOnPage('/task');**

**}**

**function createTask(AcceptanceTester $I)**

**{**

***// todo: write test***

**}**

**function viewTask(AcceptanceTester $I)**

**{**

***// todo: write test***

**}**

**function updateTask(AcceptanceTester $I)**

**{**

***// todo: write test***

**}**

**function deleteTask(AcceptanceTester $I)**

**{**

***// todo: write test***

**}**

**}**

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

## INTERACTIVE PAUSE

It’s hard to write a complete test at once. You will need to try different commands with different arguments before you find a correct path.

Since Codeception 3.0 you can pause the execution at any point and enter an interactive shell where you will be able to try different commands in action. All you need to do is **call $I->pause()** somewhere in your test, then run the test in [debug mode](https://codeception.com/docs/02-GettingStarted#Debugging).

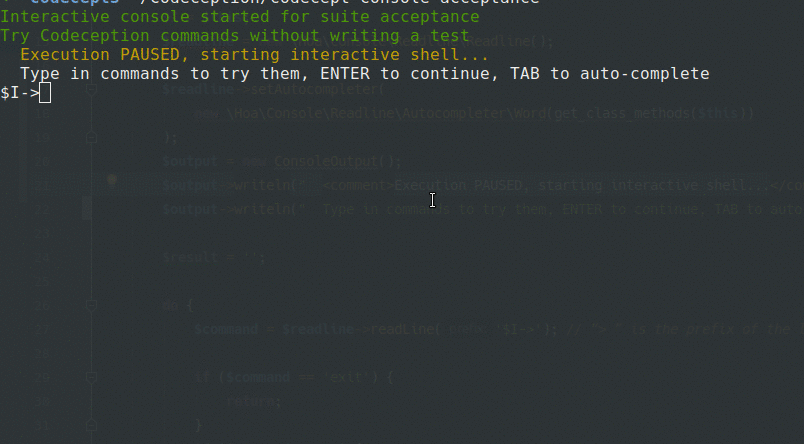
Interactive Pause requires [hoa/console](https://hoa-project.net/) which is not installed by default. To install it, run:

**php composer.phar require --dev hoa/console**

***// use pause inside a test:***

**$I->pause();**

The execution of the test is stopped at this point, and a console is shown where you can try all available commands “live”. This can be very useful when you write functional, acceptance, or api test.



Inside Interactive Pause you can use the entire power of the PHP interpreter: variables, functions, etc. You can access the result of the last executed command in a variable called $result.

In acceptance or functional test you can save page screenshot or html snapshot.

***// inside PhpBrowser, WebDrvier, frameworks***

***// saves current HTML and prints a path to created file***

**$I->makeHtmlSnapshot();**

***// inside WebDriver***

***// saves screenshot and prints a path to created file***

**$I->makeScreenshot();**

To try commands without running a single test you can launch interactive console:

**$ php vendor/bin/codecept console suitename**

Now you can execute all the commands of a corresponding Actor class and see the results immediately.

## 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](https://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 vendor/bin/codecept run**

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

**php vendor/bin/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 vendor/bin/codecept run acceptance SigninCest.php**

Alternatively you can provide the full path to test file:

**php vendor/bin/codecept run tests/acceptance/SigninCest.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 vendor/bin/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 vendor/bin/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 vendor/bin/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](https://codeception.com/docs/07-AdvancedUsage#Groups).

### REPORTS

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

**php vendor/bin/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 vendor/bin/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: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 >**](https://codeception.com/docs/03-AcceptanceTests)
* **Previous Chapter:**[**< Introduction**](https://codeception.com/docs/01-Introduction)

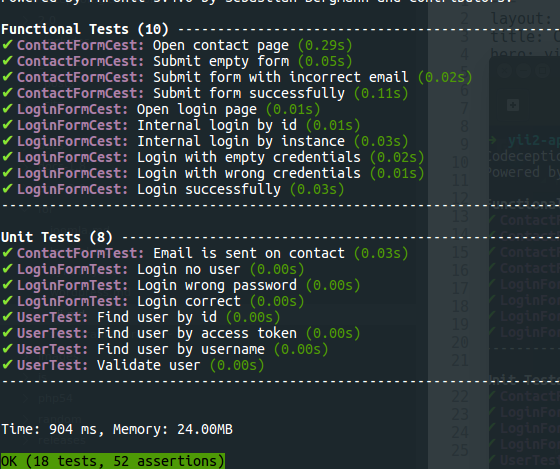
Yii Framework provides basic and advanced application templates. Both include sample Codeception tests, thus to start with Codeception you need to start new Yii project from a one of those templates.

## YII BASIC

Once you created a project from a [basic template](https://github.com/yiisoft/yii2-app-basic/) you should see folder tests and codeception.yml configuration file. From start there are functional and unit test suites.

Run them by executing in terminal:

**./vendor/bin/codecept run**



### UNIT TESTS

Unit tests are located in tests/unit directory and are supposed to contain all kind of unit and integration testing.

Each test case extends Codeception\Test\Unit class, which is standard Codeception format for unit testing. It is pretty hard to develop completely isolated unit tests in Yii, so an application is bootstrapped before each test case. Tests are configured in tests/unit.suite.yml file with Yii2 module enabled:

**modules:**

**enabled:**

**- Yii2:**

**part: [orm, email]**

This module starts Yii application for a test case and provides additional helper methods to simplify testing. It has only orm and email parts in order to exclude methods needed for functional testing only.

By accessing $this->tester class inside a test case you can use methods of [Yii2 module](https://codeception.com/docs/modules/Yii2). So if you have orm and email parts enabled so you can call methods belonging from these parts:

***// insert records in database***

**$this->tester->haveRecord('app/model/User', ['username' => 'davert']);**

***// check records in database***

**$this->tester->seeRecord('app/model/User', ['username' => 'davert']);**

***// test email was sent***

**$this->tester->seeEmailIsSent();**

***// get a last sent emails***

**$this->tester->grabLastSentEmail();**

If you enable fixtures part you will also get methods to load and use fixtures in your tests:

***// load fixtures***

**$this->tester->haveFixtures([**

**'user' => [**

**'class' => UserFixture::className(),**

***// fixture data located in tests/\_data/user.php***

**'dataFile' => codecept\_data\_dir() . 'user.php'**

**]**

**]);**

***// get first user from fixtures***

**$this->tester->grabFixture('user', 0);**

If Yii2 module is enabled you can safely call Yii::$app inside a test, as application is initialized and cleaned up after a test. If you want to add your helper methods or custom assertions for your test case you should not extend Codeception\Test\Unit but write your own separate [Helper class](https://codeception.com/docs/06-ModulesAndHelpers#Helpers).

 Continue to [Unit Testing Guide »](https://codeception.com/docs/05-UnitTests)

### FUNCTIONAL TESTS

When it comes to test real features of web applications you can’t go with unit testing only. You want to test how application handles the requests, what responses it provides, what data is saved to database and so on. To test application in near user environment but without launching real webserver or a browser you can use functional tests. They are far more simpler than unit tests in a way they are written. They describe interaction scenario in a simple DSL so you don’t need to deal with application directly but describe actions from a user’s perspective:

**$I->amOnPage(['site/contact']);**

**$I->submitForm('#contact-form', []);**

**$I->expectTo('see validations errors');**

**$I->see('Contact', 'h1');**

**$I->see('Name cannot be blank');**

**$I->see('Email cannot be blank');**

**$I->see('Subject cannot be blank');**

**$I->see('Body cannot be blank');**

This way you not only test the ContactForm on a site but actual output of application that user sees. Codeception provides standard set of actions like amOnPage, submitForm, see for testing. Yii2 module provides special methods, like amLoggedInAs (for fast authentication), haveRecord, seeRecord, seeEmailIsSent and others. They all are listed in [module reference](https://codeception.com/docs/modules/Yii2).

Functional tests should be written inside [Cest files](https://codeception.com/docs/07-AdvancedUsage#Cest-Classes), which is a scenario-driven test format of Codeception. You can easily create a new test by running:

**./vendor/bin/codecept g:cest functional MyNewScenarioCest**

 Functional tests are really powerful and simple. They are highly **recommended to use** for any Yii application.  
Continue to [Functional Testing Guide »](https://codeception.com/docs/04-FunctionalTests)

### API TESTS

API tests are not included in any Yii templates so you need to set up them manually if you developing a web service. API testing is done at functional testing level but instead of testing HTML responses on user actions, they test requests and responses via protocols like REST or SOAP. To start writing api tests you should create a suite for them

**./vendor/bin/codecept g:suite api**

You will need to enable REST, Yii2 module in tests/api.suite.yml:

**class\_name: ApiTester**

**modules:**

**enabled:**

**- REST:**

**url: /api/v1**

**depends: Yii2**

**- \ApiBundle\Helper\Api**

**config:**

**- Yii2**

Yii2 module actions like amOnPage or see should not be available for testing API. This is why Yii2 module is not enabled but declared with depends for REST module.

 Continue to [REST API Testing Guide »](https://codeception.com/docs/10-APITesting#REST-API).

### ACCEPTANCE TESTS

From a test perspective acceptance tests do the same as functional tests. They test the user interaction with application but in this case using real browser and web server. They are much slower and much more fragile. They should not duplicate functional tests in matter of testing functionality but should be used for testing the UI of your application. If you are unsure which tests should be acceptance and which are functional, write acceptance tests for JavaScript-rich applications, where UI highly depends on a browser processing. You can also use acceptance tests for happy-path scenarios, just to ensure that a real user using a real browser achieve the same results you expect in functional tests.

By default in basic application acceptance tests are disabled (as they require web server, Selenium Server and browser to be running). You can easily enable them by renaming acceptance.suite.yml.example to acceptance.suite.yml

**mv tests/acceptance.suite.yml.example tests/acceptance.suite.yml**

Basic template uses codeception/base package which doesn’t contain facebook/webdriver library required to run acceptance tests. Please change codeception/base to codeception/codeception in composer.json and run the update command.

Then you will need to launch application server in test mode:

**./tests/bin/yii serve**

and start a [Selenium Server](https://codeception.com/docs/modules/WebDriver#Local-Testing). For acceptance WebDriver module is used. Please check its reference to learn how to work with it. Unlike Yii2 module it does know nothing about your application, so if you want to use features of Yii like fixtures for acceptance testing, you should check that enable Yii2 module is enabled as well:

***# config at tests/acceptance.yml***

**modules:**

**enabled:**

**- WebDriver:**

**url: http://127.0.0.1:8080/**

**browser: firefox**

**- Yii2:**

**part: [orm, fixtures] *# allow to use AR methods***

**cleanup: false *# don't wrap test in transaction***

**entryScript: index-test.php**

As it was said, functional and acceptance tests are similar, so in order to avoid conflicts with these modules you should load only a part of Yii2 module which you really need. You must also set cleanup: false so Yii2 changes to database to be saved and used by application running on web server. Use entryScript and entryUrl values to change the default host and script configuration for your app.

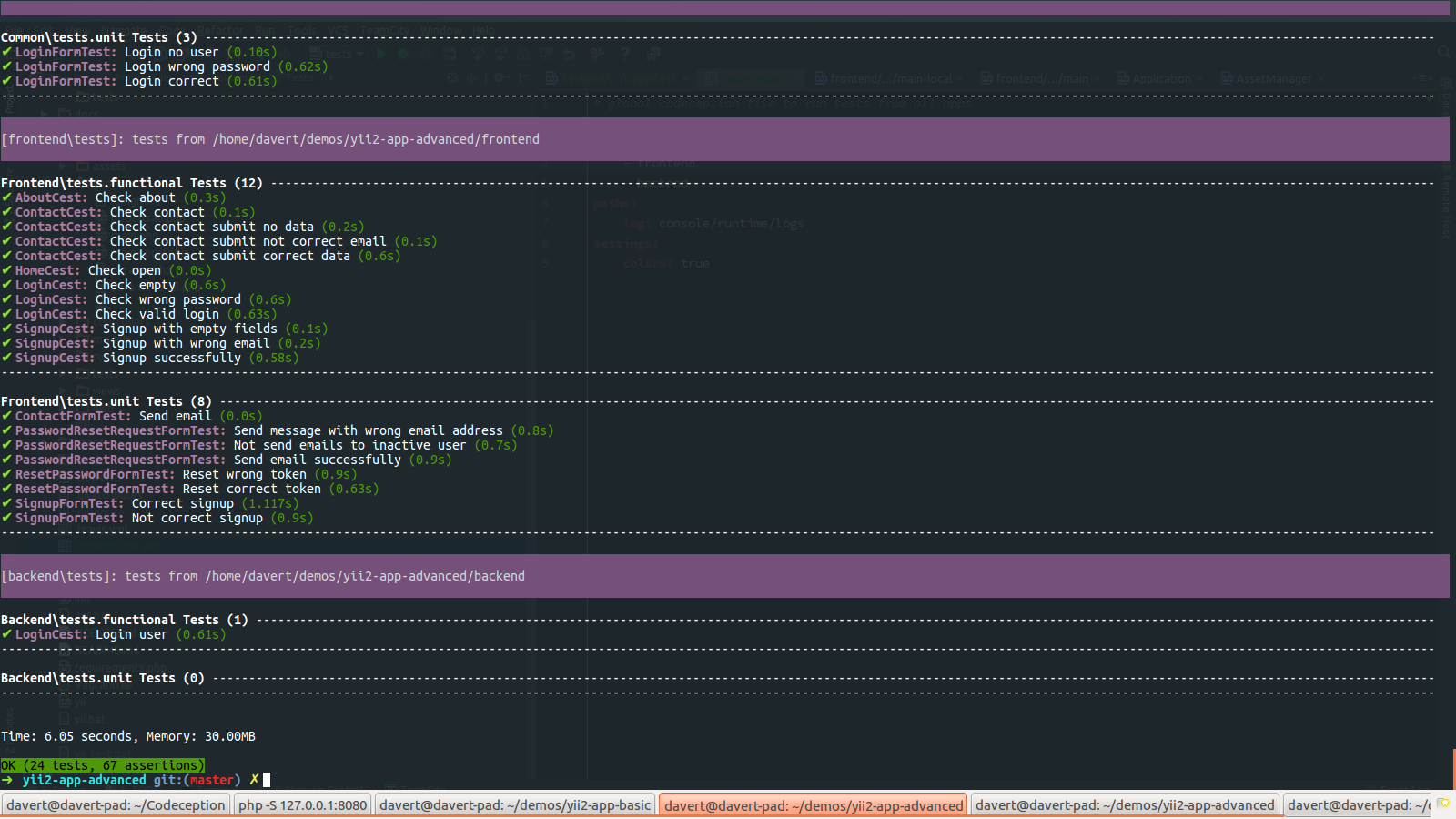
Similar as for functional tests it is recommended to use Cest format for acceptance testing:

**./vendor/bin/codecept g:cest acceptance MyNewScenarioCest**

 Continue to [Acceptance Testing Guide »](https://codeception.com/docs/03-AcceptanceTests)

## YII ADVANCED

Advanced template also has Codeception preinstalled with examples of unit, functional, and acceptance tests. However, you won’t find tests folder in a root of application. This because every application frontend, backend, and their common has their own tests. This is done to have tests and source to be placed in a same location. To run all tests from all application at once, you should execute codecept run from a project root. Global codeception.yml config was written to include tests from all applications.



Tests also include namespaces for testcase classes and testers to avoid conflicts between tests from different applications.

 Learn more about [testing multiple applications »](https://codeception.com/docs/08-Customization#One-Runner-for-Multiple-Applications)

## MANUAL SETUP && CONFIGURATION

To start you need to install Codeception via Composer

**composer require "codeception/codeception" --dev**

Create basic test suites

**./vendor/bin/codecept bootstrap**

Enable module Yii2 for functional tests inside functional.suite.yml:

***# functional.suite.yml***

**modules:**

**enabled:**

**- Yii2:**

**configFile: *#insert path to config file***

The only required parameter for Yii2 module is configFile. This file with configuration for test configuration of Yii application. It should merge original application config overriding id value and provide different database for testing:

***// config/test.php***

**$config = yii\helpers\ArrayHelper::merge(**

**require(\_\_DIR\_\_ . '/main.php'),**

**require(\_\_DIR\_\_ . '/main-local.php'),**

**[**

**'id' => 'app-tests',**

**'components' => [**

**'db' => [**

**'dsn' => 'mysql:host=localhost;dbname=yii\_app\_test',**

**]**

**]**

**]**

**);**

**return $config;**

Test config is recommended to store in config folder of application. You should provide path to test config relatively to codeception.yml file.

Please also make sure that YII\_ENV constant is set to test as it is done in tests/\_bootstrap.php file of basic and advanced app templates.

Once you configured functional tests it should be easy to create setup for unit and acceptance tests, as it is described in this guide.

In basic and advanced application templates configFile is defined in global configuration file:

***# inside codeception.yml***

**modules:**

**config:**

**Yii2:**

**configFile: 'config/test.php'**

This way we don’t need to provide test config for each defined suite.

# ACCEPTANCE TESTING

Acceptance testing can be performed by a non-technical person. That person can be your tester, manager or even client. If you are developing a web-application (and you probably are) the tester needs nothing more than a web browser to check that your site works correctly. You can reproduce an acceptance tester’s actions in scenarios and run them automatically. Codeception keeps tests clean and simple as if they were recorded from the words of an actual acceptance tester.

It makes no difference what (if any) CMS or framework is used on the site. You can even test sites created with different languages, like Java, .NET, etc. It’s always a good idea to add tests to your website. At least you will be sure that site features work after the latest changes were made.

## SAMPLE SCENARIO

Let’s say the first test you would want to run, would be signing in. In order to write such a test, we still require basic knowledge of PHP and HTML:

**$I->amOnPage('/login');**

**$I->fillField('username', 'davert');**

**$I->fillField('password', 'qwerty');**

**$I->click('LOGIN');**

**$I->see('Welcome, Davert!');**

**This scenario can be performed either by PhpBrowser or by a “real” browser through WebDriver**.

|  |  |  |
| --- | --- | --- |
|  | PhpBrowser | WebDriver |
| Browser Engine | Guzzle + Symfony BrowserKit | Chrome or Firefox |
| JavaScript | No | Yes |
| see/seeElement checks if… | …text is present in the HTML source | …text is actually visible to the user |
| Access to HTTP response headers and status codes | Yes | No |
| System requirements | PHP with [ext-curl](https://php.net/manual/book.curl.php) | Chrome or Firefox; optionally with Selenium Standalone Server |
| Speed | Fast | Slow |

We will start writing our first acceptance tests with PhpBrowser.

## PHPBROWSER

This is the fastest way to run acceptance tests since it doesn’t require running an actual browser. We use a PHP web scraper, which acts like a browser: It sends a request, then receives and parses the response. Codeception uses [Guzzle](http://guzzlephp.org/) and [Symfony BrowserKit](https://symfony.com/doc/current/components/browser_kit.html) to interact with HTML web pages.

Common PhpBrowser drawbacks:

* You can only click on links with valid URLs or form submit buttons
* You can’t fill in fields that are not inside a form

We need to specify the url parameter in the acceptance suite config:

***# acceptance.suite.yml***

**actor: AcceptanceTester**

**modules:**

**enabled:**

**- PhpBrowser:**

**url: http://www.example.com/**

**- \Helper\Acceptance**

We should start by creating a test with the next command:

**vendor/bin/codecept g:cest acceptance Signin**

It will be placed into tests/acceptance directory.

**class SigninCest**

**{**

**public function tryToTest(AcceptanceTester $I)**

**{**

**}**

**}**

The $I object is used to write all interactions. The methods of the $I object are taken from the [PhpBrowser Module](https://codeception.com/docs/modules/PhpBrowser). We will briefly describe them here:

**$I->amOnPage('/login');**

We will assume that all actions starting with am and have describe the initial environment. The amOnPage action sets the starting point of a test to the /login page.

With the PhpBrowser you can click the links and fill in the forms. That will probably be the majority of your actions.

#### Click

Emulates a click on valid anchors. The URL referenced in the href attribute will be opened. As a parameter, you can specify the link name or a valid CSS or XPath selector.

**$I->click('Log in');**

***// CSS selector applied***

**$I->click('#login a');**

***// XPath***

**$I->click('//a[@id=login]');**

***// Using context as second argument***

**$I->click('Login', '.nav');**

Codeception tries to locate an element by its text, name, CSS or XPath. You can specify the locator type manually by passing an array as a parameter. We call this a **strict locator**. Available strict locator types are:

* id
* name
* css
* xpath
* link
* class

***// By specifying locator type***

**$I->click(['link' => 'Login']);**

**$I->click(['class' => 'btn']);**

There is a special class [Codeception\Util\Locator](https://codeception.com/docs/reference/Locator) which may help you to generate complex XPath locators. For instance, it can easily allow you to click an element on the last row of a table:

**$I->click('Edit' , \Codeception\Util\Locator::elementAt('//table/tr', -1));**

#### Forms

Clicking links is probably not what takes the most time during the testing of a website. The most routine waste of time goes into the testing of forms. Codeception provides several ways of testing forms.

Let’s submit this sample form inside the Codeception test:

**<form method="post" action="/update" id="update\_form">**

**<label for="user\_name">Name</label>**

**<input type="text" name="user[name]" id="user\_name" />**

**<label for="user\_email">Email</label>**

**<input type="text" name="user[email]" id="user\_email" />**

**<label for="user\_gender">Gender</label>**

**<select id="user\_gender" name="user[gender]">**

**<option value="m">Male</option>**

**<option value="f">Female</option>**

**</select>**

**<input type="submit" name="submitButton" value="Update" />**

**</form>**

From a user’s perspective, a form consists of fields which should be filled in, and then a submit button clicked:

***// we are using label to match user\_name field***

**$I->fillField('Name', 'Miles');**

***// we can use input name or id***

**$I->fillField('user[email]','miles@davis.com');**

**$I->selectOption('Gender','Male');**

**$I->click('Update');**

To match fields by their labels, you should write a for attribute in the label tag.

From the developer’s perspective, submitting a form is just sending a valid POST request to the server. Sometimes it’s easier to fill in all of the fields at once and send the form without clicking a ‘Submit’ button. A similar scenario can be rewritten with only one command:

**$I->submitForm('#update\_form', array('user' => array(**

**'name' => 'Miles',**

**'email' => 'Davis',**

**'gender' => 'm'**

**)));**

The submitForm is not emulating a user’s actions, but it’s quite useful in situations when the form is not formatted properly, for example, to discover that labels aren’t set or that fields have unclean names or badly written IDs, or the form is sent by a JavaScript call.

By default, submitForm doesn’t send values for buttons. The last parameter allows specifying what button values should be sent, or button values can be explicitly specified in the second parameter:

**$I->submitForm('#update\_form', array('user' => array(**

**'name' => 'Miles',**

**'email' => 'Davis',**

**'gender' => 'm'**

**)), 'submitButton');**

***// this would have the same effect, but the value has to be explicitly specified***

**$I->submitForm('#update\_form', array('user' => array(**

**'name' => 'Miles',**

**'email' => 'Davis',**

**'gender' => 'm',**

**'submitButton' => 'Update'**

**)));**

##### Hiding Sensitive Data

If you need to fill in sensitive data (like passwords) and hide it in logs, you can pass instance \Codeception\Step\Argument\PasswordArgument with the data which needs to be hidden.

**use \Codeception\Step\Argument\PasswordArgument;**

**$I->amOnPage('/form/password\_argument');**

**$I->fillField('password', new PasswordArgument('thisissecret'));**

thisissecret will be filled into a form but it won’t be shown in output and logs.

#### Assertions

In the PhpBrowser you can test the page contents. In most cases, you just need to check that the required text or element is on the page.

The most useful method for this is see():

***// We check that 'Thank you, Miles' is on the page.***

**$I->see('Thank you, Miles');**

***// We check that 'Thank you, Miles' is inside an element with 'notice' class.***

**$I->see('Thank you, Miles', '.notice');**

***// Or using XPath locators***

**$I->see('Thank you, Miles', "//table/tr[2]");**

***// We check this message is \*not\* on the page.***

**$I->dontSee('Form is filled incorrectly');**

You can check that a specific HTML element exists (or doesn’t) on a page:

**$I->seeElement('.notice');**

**$I->dontSeeElement('.error');**

We also have other useful commands to perform checks. Please note that they all start with the see prefix:

**$I->seeInCurrentUrl('/user/miles');**

**$I->seeCheckboxIsChecked('#agree');**

**$I->seeInField('user[name]', 'Miles');**

**$I->seeLink('Login');**

#### Conditional Assertions

Usually, as soon as any assertion fails, further assertions of this test will be skipped. Sometimes you don’t want this - maybe you have a long-running test and you want it to run to the end. In this case, you can use conditional assertions. Each see method has a corresponding canSee method, and dontSee has a cantSee method:

**$I->canSeeInCurrentUrl('/user/miles');**

**$I->canSeeCheckboxIsChecked('#agree');**

**$I->cantSeeInField('user[name]', 'Miles');**

Each failed assertion will be shown in the test results, but it won’t stop the test.

Conditional assertions are disabled in bootstrap setup. To enable them you should add corresponding step decorators to suite config:

If you started project as codecept init acceptance they should be already enabled in config

***# in acceptance.suite.yml***

***# or in codeception.yml inside suites section***

**step\_decorators:**

**- \Codeception\Step\ConditionalAssertion**

Then rebuild actors with codecept build command.

#### Comments

Within a long scenario, you should describe what actions you are going to perform and what results should be achieved. Comment methods like amGoingTo, expect, expectTo help you in making tests more descriptive:

**$I->amGoingTo('submit user form with invalid values');**

**$I->fillField('user[email]', 'miles');**

**$I->click('Update');**

**$I->expect('the form is not submitted');**

**$I->see('Form is filled incorrectly');**

#### Grabbers

These commands retrieve data that can be used in the test. Imagine your site generates a password for every user and you want to check that the user can log into the site using this password:

**$I->fillField('email', 'miles@davis.com');**

**$I->click('Generate Password');**

**$password = $I->grabTextFrom('#password');**

**$I->click('Login');**

**$I->fillField('email', 'miles@davis.com');**

**$I->fillField('password', $password);**

**$I->click('Log in!');**

Grabbers allow you to get a single value from the current page with commands:

**$token = $I->grabTextFrom('.token');**

**$password = $I->grabTextFrom("descendant::input/descendant::\*[@id = 'password']");**

**$api\_key = $I->grabValueFrom('input[name=api]');**

#### Cookies, URLs, Title, etc

Actions for cookies:

**$I->setCookie('auth', '123345');**

**$I->grabCookie('auth');**

**$I->seeCookie('auth');**

Actions for checking the page title:

**$I->seeInTitle('Login');**

**$I->dontSeeInTitle('Register');**

Actions for URLs:

**$I->seeCurrentUrlEquals('/login');**

**$I->seeCurrentUrlMatches('~^/users/(\d+)~');**

**$I->seeInCurrentUrl('user/1');**

**$user\_id = $I->grabFromCurrentUrl('~^/user/(\d+)/~');**

## WEBDRIVER

A nice feature of Codeception is that most scenarios are similar, no matter of how they are executed. PhpBrowser was emulating browser requests but how to execute such test in a real browser like Chrome or Firefox? Selenium WebDriver can drive them so in our acceptance tests we can automate scenarios we used to test manually. In such tests, we should concentrate more on **testing the UI** than on testing functionality.

“[WebDriver](https://www.w3.org/TR/webdriver/)” is the name of a protocol (specified by W3C) to drive browsers automatically. This specification is implemented for all modern desktop and mobile browsers. Codeception uses [php-webdriver/php-webdriver](https://github.com/php-webdriver/php-webdriver) as a PHP implementation of the WebDriver protocol.

To control the browsers you need to use a program or a service to start/stop browser sessions. In the next section, we will overview the most popular solutions.

### LOCAL SETUP

#### Selenium Server

[Selenium Server](http://www.seleniumhq.org/) is a de-facto standard for automated web and mobile testing. It is a server that can launch and drive different browsers locally or remotely. WebDriver protocol was initially created by Selenium before becoming a W3C standard. This makes Selenium server the most stable complete implementation of WebDriver for today. Selenium Server is also recommended by Codeception team.

To control browsers Selenium Server uses official tools maintained by browser vendors, like [ChromeDriver](https://sites.google.com/a/chromium.org/chromedriver) for Chrome or [GeckoDriver](https://github.com/mozilla/geckodriver) for Firefox. This makes Selenium quite heavy to install, as it requires Java, browsers, Chrome or GeckoDriver and GUI (display server) to run browsers in.

* Follow [Installation Instructions](https://codeception.com/docs/modules/WebDriver#Selenium)
* Enable [RunProcess](https://codeception.com/extensions#RunProcess) extension to start/stop Selenium automatically (optional).

#### ChromeDriver

ChromeDriver was created by Google to control Chrome and Chromium browsers programmatically. It can be paired with [Selenium Server](https://codeception.com/docs/03-AcceptanceTests#Selenium-Server) or used as a standalone tool to drive Chrome browser. It is simpler to set up than Selenium Server, however, it has limited support for WebDriver protocol.

* Follow [Installation Instructions](https://codeception.com/docs/modules/WebDriver#ChromeDriver)
* Enable [RunProcess](https://codeception.com/extensions#RunProcess) extension to start/stop ChromeDriver automatically (optional).

### CONFIGURATION

To execute a test in a browser we need to change the suite configuration to use **WebDriver** instead of PhpBrowser.

Modify your acceptance.suite.yml file:

**actor: AcceptanceTester**

**modules:**

**enabled:**

**- WebDriver:**

**url:**

**browser: chrome**

**- \Helper\Acceptance**

See [WebDriver Module](https://codeception.com/docs/modules/WebDriver) for details.

Please note that actions executed in a browser will behave differently. For instance, seeElement won’t just check that the element exists on a page, but it will also check that element is actually visible to the user:

**$I->seeElement('#modal');**

While WebDriver duplicates the functionality of PhpBrowser, it has its limitations: It can’t check headers since browsers don’t provide APIs for that. WebDriver also adds browser-specific functionality:

#### Wait

While testing web application, you may need to wait for JavaScript events to occur. Due to its asynchronous nature, complex JavaScript interactions are hard to test. That’s why you may need to use waiters, actions with wait prefix. They can be used to specify what event you expect to occur on a page, before continuing the test.

For example:

**$I->waitForElement('#agree\_button', 30); *// secs***

**$I->click('#agree\_button');**

In this case, we are waiting for the ‘agree’ button to appear and then click it. If it didn’t appear after 30 seconds, the test will fail. There are other wait methods you may use, like [waitForText](https://codeception.com/docs/modules/WebDriver#waitForText), [waitForElementVisible](https://codeception.com/docs/modules/WebDriver#waitForElementVisible) and others.

If you don’t know what exact element you need to wait for, you can simply pause execution with using $I->wait()

**$I->wait(3); *// wait for 3 secs***

#### SmartWait

It is possible to wait for elements pragmatically. If a test uses element which is not on a page yet, Codeception will wait for few extra seconds before failing. This feature is based on [Implicit Wait](http://www.seleniumhq.org/docs/04_webdriver_advanced.jsp#implicit-waits) of Selenium. Codeception enables implicit wait only when searching for a specific element and disables in all other cases. Thus, the performance of a test is not affected.

SmartWait can be enabled by setting wait option in WebDriver config. It expects the number of seconds to wait. Example:

**wait: 5**

With this config we have the following test:

***// we use wait: 5 instead of***

***// $I->waitForElement(['css' => '#click-me'], 5);***

***// to wait for element on page***

**$I->click(['css' => '#click-me']);**

It is important to understand that SmartWait works only with a specific locators:

* #locator - CSS ID locator, works
* //locator - general XPath locator, works
* ['css' => 'button''] - strict locator, works

But it won’t be executed for all other locator types. See the example:

**$I->click('Login'); *// DISABLED, not a specific locator***

**$I->fillField('user', 'davert'); *// DISABLED, not a specific locator***

**$I->fillField(['name' => 'password'], '123456'); *// ENABLED, strict locator***

**$I->click('#login'); *// ENABLED, locator is CSS ID***

**$I->see('Hello, Davert'); *// DISABLED, Not a locator***

**$I->seeElement('#userbar'); *// ENABLED***

**$I->dontSeeElement('#login'); *// DISABLED, can't wait for element to hide***

**$I->seeNumberOfElements(['css' => 'button.link'], 5); *// DISABLED, can wait only for one element***

#### Retry

When it’s hard to define condition to wait for, we can retry a command few times until it succeeds. For instance, if you try to click while it’s animating you can try to do it few times until it freezes. Since Codeception 3.0 each action and assertion have an alias prefixed with retry which allows to retry a flaky command.

**$I->retryClick('flaky element');**

**$I->retrySee('Something changed');**

Retry can be configured via $I->retry() command, where you can set number of retries and initial interval: interval will be doubled on each unsuccessful execution.

***// Retry up to 6 sec: 4 times, for 400ms initial interval => 400ms + 800ms + 1600ms + 3200ms = 6000ms***

**$I->retry(4, 400);**

$I->retry takes 2 parameters:

* number of retries (1 by default)
* initial interval (200ms by default)

Retries are disabled by default. To enable them you should add retry step decorators to suite config:

If you started project as codecept init acceptance they should be already enabled in config

***# in acceptance.suite.yml***

***# or in codeception.yml inside suites section***

**step\_decorators:**

**- \Codeception\Step\Retry**

Then add \Codeception\Lib\Actor\Shared\Retry trait into AcceptanceTester class:

**class AcceptanceTester extends \Codeception\Actor**

**{**

**use \_generated\AcceptanceTesterActions;**

**use \Codeception\Lib\Actor\Shared\Retry;**

**}**

Run codecept build to recreate actions. New retry\* actions are available for tests. Keep in mind, that you can change retry policy dynamically for each test.

#### Wait and Act

To combine waitForElement with actions inside that element you can use the [performOn](https://codeception.com/docs/modules/WebDriver#performOn) method. Let’s see how you can perform some actions inside an HTML popup:

**$I->performOn('.confirm', \Codeception\Util\ActionSequence::build()**

**->see('Warning')**

**->see('Are you sure you want to delete this?')**

**->click('Yes')**

**);**

Alternatively, this can be executed using a callback, in this case the WebDriver instance is passed as argument

**$I->performOn('.confirm', function(\Codeception\Module\WebDriver $I) {**

**$I->see('Warning');**

**$I->see('Are you sure you want to delete this?');**

**$I->click('Yes');**

**});**

For more options see [performOn() reference](https://codeception.com/docs/modules/WebDriver#performOn).

#### A/B Testing

When a web site acts unpredictably you may need to react on that change. This happens if site configured for A/B testing, or shows different popups, based on environment.

Since Codeception 3.0 you can have some actions to fail silently, if they are errored. Let’s say, you open a page and some times there is a popup which should be closed. We may try to hit the “close” button but if this action fails (no popup on page) we just continue the test.

This is how it can be implemented:

**$I->amOnPage('/');**

**$I->tryToClick('x', '.alert');**

***// continue execution***

You can also use tryTo as condition for your tests:

**if ($I->tryToSeeElement('.alert')) {**

**$I->waitForText('Do you accept cookies?');**

**$I->click('Yes');**

**}**

A/B testing is disabled by default. To enable it you should add corresponding step decorators to suite config:

If you started project as codecept init acceptance in Codeception >= 3.0 they should be already enabled in config

***# in acceptance.suite.yml***

***# or in codeception.yml inside suites section***

**step\_decorators:**

**- \Codeception\Step\TryTo**

Then rebuild actors with codecept build command.

### MULTI SESSION TESTING

Codeception allows you to execute actions in concurrent sessions. The most obvious case for this is testing realtime messaging between users on a site. In order to do it, you will need to launch two browser windows at the same time for the same test. Codeception has a very smart concept for doing this. It is called **Friends**:

**$I->amOnPage('/messages');**

**$nick = $I->haveFriend('nick');**

**$nick->does(function(AcceptanceTester $I) {**

**$I->amOnPage('/messages/new');**

**$I->fillField('body', 'Hello all!');**

**$I->click('Send');**

**$I->see('Hello all!', '.message');**

**});**

**$I->wait(3);**

**$I->see('Hello all!', '.message');**

In this case, we performed, or ‘did’, some actions in the second window with the does method on a friend object.

Sometimes you may want to close a webpage before the end of the test. For such cases, you may use leave(). You can also specify roles for a friend:

**$nickAdmin = $I->haveFriend('nickAdmin', adminStep::class);**

**$nickAdmin->does(function(adminStep $I) {**

***// Admin does ...***

**});**

**$nickAdmin->leave();**

Multi session testing is disabled by default. To enable it, add \Codeception\Lib\Actor\Shared\Friend into AcceptanceTester.

**class AcceptanceTester extends \Codeception\Actor**

**{**

**use \_generated\AcceptanceTesterActions;**

**use \Codeception\Lib\Actor\Shared\Friend;**

**}**

### CLOUD TESTING

Some environments are hard to be reproduced manually, testing Internet Explorer 6-8 on Windows XP may be a hard thing, especially if you don’t have Windows XP installed. This is where Cloud Testing services come to help you. Services such as [SauceLabs](https://saucelabs.com/), [BrowserStack](https://www.browserstack.com/) and [others](https://codeception.com/docs/modules/WebDriver#Cloud-Testing) can create virtual machines on demand and set up Selenium Server and the desired browser. Tests are executed on a remote machine in a cloud, to access local files cloud testing services provide a special application called **Tunnel**. Tunnel operates on a secured protocol and allows browsers executed in a cloud to connect to a local web server.

Cloud Testing services work with the standard WebDriver protocol. This makes setting up cloud testing really easy. You just need to set the [WebDriver configuration](https://codeception.com/docs/modules/WebDriver#Cloud-Testing) to:

* specify the host to connect to (depends on the cloud provider)
* authentication details (to use your account)
* browser
* OS

We recommend using [params](https://codeception.com/docs/06-ModulesAndHelpers#Dynamic-Configuration-With-Params) to provide authorization credentials.

It should be mentioned that Cloud Testing services are not free. You should investigate their pricing models and choose one that fits your needs. They also may work painfully slowly if ping times between the local server and the cloud is too high. This may lead to random failures in acceptance tests.

### DEBUGGING

Codeception modules can print valuable information while running. Just execute tests with the --debug option to see running details. For any custom output use the codecept\_debug function:

**codecept\_debug($I->grabTextFrom('#name'));**

On each failure, the snapshot of the last shown page will be stored in the tests/\_output directory. PhpBrowser will store the HTML code and WebDriver will save a screenshot of the page.

Additional debugging features by Codeception:

* [Interactive Pause](https://codeception.com/docs/02-GettingStarted#Interactive-Pause) is a REPL that allows to type and check commands for instant feedback.
* [Recorder Extension](https://codeception.com/addons#CodeceptionExtensionRecorder) allows to record tests step-by-steps and show them in slideshow

### COMMON CASES

Let’s see how common problems of acceptance testing can be solved with Codeception.

#### Login

It is recommended to put widely used actions inside an Actor class. A good example is the login action which would probably be actively involved in acceptance or functional testing:

**class AcceptanceTester extends \Codeception\Actor**

**{**

***// do not ever remove this line!***

**use \_generated\AcceptanceTesterActions;**

**public function login($name, $password)**

**{**

**$I = $this;**

**$I->amOnPage('/login');**

**$I->submitForm('#loginForm', [**

**'login' => $name,**

**'password' => $password**

**]);**

**$I->see($name, '.navbar');**

**}**

**}**

Now you can use the login method inside your tests:

***// $I is AcceptanceTester***

**$I->login('miles', '123456');**

However, implementing all actions for reuse in a single actor class may lead to breaking the [Single Responsibility Principle](https://en.wikipedia.org/wiki/Single_responsibility_principle).

#### Single Login

If you need to authorize a user for each test, you can do so by submitting the login form at the beginning of every test. Running those steps takes time, and in the case of Selenium tests (which are slow by themselves) that time loss can become significant.

Codeception allows you to share cookies between tests, so a test user can stay logged in for other tests.

Let’s improve the code of our login method, executing the form submission only once and restoring the session from cookies for each subsequent login function call:

**public function login($name, $password)**

**{**

**$I = $this;**

***// if snapshot exists - skipping login***

**if ($I->loadSessionSnapshot('login')) {**

**return;**

**}**

***// logging in***

**$I->amOnPage('/login');**

**$I->submitForm('#loginForm', [**

**'login' => $name,**

**'password' => $password**

**]);**

**$I->see($name, '.navbar');**

***// saving snapshot***

**$I->saveSessionSnapshot('login');**

**}**

Note that session restoration only works for WebDriver modules (modules implementing Codeception\Lib\Interfaces\SessionSnapshot).

### CUSTOM BROWSER SESSIONS

By default, WebDriver module is configured to automatically start browser before the test and stop afterward. However, this can be switched off with start: false module configuration. To start a browser you will need to write corresponding methods in Acceptance [Helper](https://codeception.com/docs/06-ModulesAndHelpers#Helpers).

WebDriver module provides advanced methods for the browser session, however, they can only be used from Helpers.

* [\_initializeSession](https://codeception.com/docs/modules/WebDriver#_initializeSession) - starts a new browser session
* [\_closeSession](https://codeception.com/docs/modules/WebDriver#_closeSession) - stops the browser session
* [\_restart](https://codeception.com/docs/modules/WebDriver#_restart) - updates configuration and restarts browser
* [\_capabilities](https://codeception.com/docs/modules/WebDriver#_capabilities) - set [desired capabilities](https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities) programmatically.

Those methods can be used to create custom commands like $I->startBrowser() or used in [before/after](https://codeception.com/docs/06-ModulesAndHelpers#Hooks) hooks.

## CONCLUSION

Writing acceptance tests with Codeception and PhpBrowser is a good start. You can easily test your Joomla, Drupal, WordPress sites, as well as those made with frameworks. Writing acceptance tests is like describing a tester’s actions in PHP. They are quite readable and very easy to write. If you need to access the database, you can use the [Db Module](https://codeception.com/docs/modules/Db).

* **Next Chapter:**[**FunctionalTests >**](https://codeception.com/docs/04-FunctionalTests)
* **Previous Chapter:**[**< GettingStarted**](https://codeception.com/docs/02-GettingStarted)

# FUNCTIONAL TESTS

Now that we’ve written some acceptance tests, functional tests are almost the same, with one major difference: Functional tests don’t require a web server.

Under the hood, Codeception uses Symfony’s [BrowserKit](https://symfony.com/doc/current/components/browser_kit.html) to “send” requests to your app. So there’s no real HTTP request made, but rather a BrowserKit [Request object](https://github.com/symfony/browser-kit/blob/master/Request.php) with the required properties is passed to your framework’s (font-)controller.

As a first step, you need to enable Codeception’s module for your framework in functional.suite.yml (see below).

All of Codeception’s framework modules share the same interface, and thus your tests are not bound to any one of them. This is a sample functional test:

***// LoginCest.php***

**class LoginCest**

**{**

**public function tryLogin(FunctionalTester $I)**

**{**

**$I->amOnPage('/');**

**$I->click('Login');**

**$I->fillField('Username', 'Miles');**

**$I->fillField('Password', 'Davis');**

**$I->click('Enter');**

**$I->see('Hello, Miles', 'h1');**

***// $I->seeEmailIsSent(); // only for Symfony***

**}**

**}**

As you see, the syntax is the same for functional and acceptance tests.

## LIMITATIONS

Functional tests are usually much faster than acceptance tests. But functional tests are less stable as they run Codeception and the application in one environment. If your application was not designed to run in long lived processes (e.g. if you use the exit operator or global variables), then functional tests are probably not for you.

### HEADERS, COOKIES, SESSIONS

One of the common issues with functional tests is the use of PHP functions that deal with headers, sessions and cookies. As you may already know, the header function triggers an error if it is executed after PHP has already output something. In functional tests we run the application multiple times, thus we will get lots of irrelevant errors in the result.

### EXTERNAL URLS

Functional tests cannot access external URLs, just URLs within your project. You can use PhpBrowser to open external URLs.

### SHARED MEMORY

In functional testing, unlike running the application the traditional way, the PHP application does not stop after it has finished processing a request. Since all requests are run in one memory container, they are not isolated. So **if you see that your tests are mysteriously failing when they shouldn’t - try to execute a single test.** This will show if the tests were failing because they weren’t isolated during the run. Keep your memory clean, avoid memory leaks and clean global and static variables.

## ENABLING FRAMEWORK MODULES

You have a functional testing suite in the tests/functional directory. To start, you need to include one of the framework modules in the suite configuration file: tests/functional.suite.yml.

### SYMFONY

To perform Symfony integration you just need to include the Symfony module into your test suite. If you also use Doctrine2, don’t forget to include it too. To make the Doctrine2 module connect using the doctrine service from Symfony, you should specify the Symfony module as a dependency for Doctrine2:

***# functional.suite.yml***

**actor: FunctionalTester**

**modules:**

**enabled:**

**- Symfony**

**- Doctrine2:**

**depends: Symfony *# connect to Symfony***

By default this module will search for AppKernel in the app directory.

The module uses the Symfony Profiler to provide additional information and assertions.

[See the full reference](https://codeception.com/docs/modules/Symfony)

### LARAVEL5

The [Laravel5](https://codeception.com/docs/modules/Laravel5) module is included and requires no configuration:

***# functional.suite.yml***

**actor: FunctionalTester**

**modules:**

**enabled:**

**- Laravel5**

### YII2

Yii2 tests are included in [Basic](https://github.com/yiisoft/yii2-app-basic) and [Advanced](https://github.com/yiisoft/yii2-app-advanced) application templates. Follow the Yii2 guides to start.

### ZEND FRAMEWORK 2

Use [the ZF2 module](https://codeception.com/docs/modules/ZF2) to run functional tests inside Zend Framework 2:

***# functional.suite.yml***

**actor: FunctionalTester**

**modules:**

**enabled:**

**- ZF2**

### ZEND EXPRESSIVE

[Zend Expressive](https://codeception.com/docs/modules/ZendExpressive) tests can be executed with enabling a corresponding module.

***# functional.suite.yml***

**actor: FunctionalTester**

**modules:**

**enabled:**

**- ZendExpressive**

See module reference to more configuration options

### PHALCON 4

The Phalcon4 module requires creating a bootstrap file which returns an instance of \Phalcon\Mvc\Application. To start writing functional tests with Phalcon support you should enable the Phalcon4 module and provide the path to this bootstrap file:

***# functional.suite.yml***

**actor: FunctionalTester**

**modules:**

**enabled:**

**- Phalcon4:**

**bootstrap: 'app/config/bootstrap.php'**

**cleanup: true**

**savepoints: true**

[See the full reference](https://codeception.com/docs/modules/Phalcon4)

## WRITING FUNCTIONAL TESTS

Functional tests are written in the same manner as [Acceptance Tests](https://codeception.com/docs/03-AcceptanceTests) with the PhpBrowser module enabled. All framework modules and the PhpBrowser module share the same methods and the same engine.

Therefore we can open a web page with amOnPage method:

**$I->amOnPage('/login');**

We can click links to open web pages:

**$I->click('Logout');**

***// click link inside .nav element***

**$I->click('Logout', '.nav');**

***// click by CSS***

**$I->click('a.logout');**

***// click with strict locator***

**$I->click(['class' => 'logout']);**

We can submit forms as well:

**$I->submitForm('form#login', ['name' => 'john', 'password' => '123456']);**

***// alternatively***

**$I->fillField('#login input[name=name]', 'john');**

**$I->fillField('#login input[name=password]', '123456');**

**$I->click('Submit', '#login');**

And do assertions:

**$I->see('Welcome, john');**

**$I->see('Logged in successfully', '.notice');**

**$I->seeCurrentUrlEquals('/profile/john');**

Framework modules also contain additional methods to access framework internals. For instance, Laravel5, Phalcon, and Yii2 modules have a seeRecord method which uses the ActiveRecord layer to check that a record exists in the database.

Take a look at the complete reference for the module you are using. Most of its methods are common to all modules but some of them are unique.

You can also access framework globals inside a test or access the dependency injection container inside the Helper\Functional class:

**namespace Helper;**

**class Functional extends \Codeception\Module**

**{**

**function doSomethingWithMyService()**

**{**

**$service = $this->getModule('Symfony')->grabServiceFromContainer('myservice');**

**$service->doSomething();**

**}**

**}**

Also check all available Public Properties of the used modules to get full access to their data.

## ERROR REPORTING

By default Codeception uses the E\_ALL & ~E\_STRICT & ~E\_DEPRECATED error reporting level. In functional tests you might want to change this level depending on your framework’s error policy. The error reporting level can be set in the suite configuration file:

**actor: FunctionalTester**

**modules:**

**enabled:**

**- Yii2**

**- \Helper\Functional**

**error\_level: "E\_ALL & ~E\_STRICT & ~E\_DEPRECATED"**

error\_level can also be set globally in codeception.yml file.

## CONCLUSION

Functional tests are great if you are using powerful frameworks. By using functional tests you can access and manipulate their internal state. This makes your tests shorter and faster. In other cases, if you don’t use frameworks there is no practical reason to write functional tests. If you are using a framework other than the ones listed here, create a module for it and share it with the community.

* **Next Chapter:**[**UnitTests >**](https://codeception.com/docs/05-UnitTests)
* **Previous Chapter:**[**< AcceptanceTests**](https://codeception.com/docs/03-AcceptanceTests)

# UNIT & INTEGRATION 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.

## CREATING A TEST

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

**php vendor/bin/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 vendor/bin/codecept run unit ExampleTest**

Or simply run the whole set of unit tests with:

**php vendor/bin/codecept run unit**

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

**class ExampleTest extends \Codeception\Test\Unit**

**{**

**/\*\***

**\* @var \UnitTester**

**\*/**

**protected $tester;**

**protected function \_before()**

**{**

**}**

**protected function \_after()**

**{**

**}**

***// tests***

**public function testMe()**

**{**

**}**

**}**

Inside a class:

* all public methods with test prefix are tests
* \_before method is executed before each test (like setUp in PHPUnit)
* \_after method is executed after each test (like tearDown in PHPUnit)

## UNIT TESTING

Unit tests are focused around a single component of an application. All external dependencies for components should be replaced with test doubles.

A typical unit test may look like this:

**class UserTest extends \Codeception\Test\Unit**

**{**

**public function testValidation()**

**{**

**$user = new User();**

**$user->setName(null);**

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

**$user->setName('toolooooongnaaaaaaameeee');**

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

**$user->setName('davert');**

**$this->assertTrue($user->validate(['username']));**

**}**

**}**

### ASSERTIONS

There are pretty many assertions you can use inside tests. The most common are:

* $this->assertEquals()
* $this->assertContains()
* $this->assertFalse()
* $this->assertTrue()
* $this->assertNull()
* $this->assertEmpty()

Assertion methods come from PHPUnit. [See the complete reference at phpunit.de](https://phpunit.de/manual/current/en/appendixes.assertions.html).

### TEST DOUBLES

Codeception provides [Codeception\Stub library](https://github.com/Codeception/Stub) for building mocks and stubs for tests. Under the hood it used PHPUnit’s mock builder but with much simplified API.

Alternatively, [Mockery](https://github.com/Codeception/MockeryModule) can be used inside Codeception.

#### Stubs

Stubs can be created with a static methods of Codeception\Stub.

**$user = \Codeception\Stub::make('User', ['getName' => 'john']);**

**$name = $user->getName(); *// 'john'***

[See complete reference](https://codeception.com/docs/reference/Mock)

Inside unit tests (Codeception\Test\Unit) it is recommended to use alternative API:

***// create a stub with find method replaced***

**$userRepository = $this->make(UserRepository::class, ['find' => new User]);**

**$userRepository->find(1); *// => User***

***// create a dummy***

**$userRepository = $this->makeEmpty(UserRepository::class);**

***// create a stub with all methods replaced except one***

**$user = $this->makeEmptyExcept(User::class, 'validate');**

**$user->validate($data);**

***// create a stub by calling constructor and replacing a method***

**$user = $this->construct(User::class, ['name' => 'davert'], ['save' => false]);**

***// create a stub by calling constructor with empty methods***

**$user = $this->constructEmpty(User::class, ['name' => 'davert']);**

***// create a stub by calling constructor with empty methods***

**$user = $this->constructEmptyExcept(User::class, 'getName', ['name' => 'davert']);**

**$user->getName(); *// => davert***

**$user->setName('jane'); *// => this method is empty***

[See complete reference](https://codeception.com/docs/reference/Mock)

Stubs can also be created using static methods from Codeception\Stub class. In this

**\Codeception\Stub::make(UserRepository::class, ['find' => new User]);**

See a reference for [static Stub API](https://codeception.com/docs/reference/Stub)

#### Mocks

To declare expectations for mocks use Codeception\Stub\Expected class:

***// create a mock where $user->getName() should never be called***

**$user = $this->make('User', [**

**'getName' => Expected::never(),**

**'someMethod' => function() {}**

**]);**

**$user->someMethod();**

***// create a mock where $user->getName() should be called at least once***

**$user = $this->make('User', [**

**'getName' => Expected::atLeastOnce('Davert')**

**]**

**);**

**$user->getName();**

**$userName = $user->getName();**

**$this->assertEquals('Davert', $userName);**

[See complete reference](https://codeception.com/docs/reference/Mock)

## INTEGRATION TESTS

Unlike unit tests integration tests doesn’t require the code to be executed in isolation. That allows us to use database and other components inside a tests. To improve the testing experience modules can be used as in functional testing.

### 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:

**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](https://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:

**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:

**function testUserNameCanBeChanged()**

**{**

***// create a user from framework, user will be deleted after the test***

**$id = $this->tester->haveInRepository(User::class, ['name' => 'miles']);**

***// get entity manager by accessing module***

**$em = $this->getModule('Doctrine2')->em;**

***// get real user***

**$user = $em->find(User::class, $id);**

**$user->setName('bill');**

**$em->persist($user);**

**$em->flush();**

**$this->assertEquals('bill', $user->getName());**

***// verify data was saved using framework methods***

**$this->tester->seeInRepository(User::class, ['name' => 'bill']);**

**$this->tester->dontSeeInRepository(User::class, ['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:

**/\*\* @var Doctrine\ORM\EntityManager \*/**

**$em = $this->getModule('Doctrine2')->em;**

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

**/\*\* @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.

### SCENARIO DRIVEN TESTING

[Cest format](https://codeception.com/docs/07-AdvancedUsage#Cest-Classes) can also be used for integration testing. In some cases it makes tests cleaner as it simplifies module access by using common $I-> syntax:

**public function buildShouldHaveSequence(\UnitTester $I)**

**{**

**$build = $I->have(Build::class, ['project\_id' => $this->project->id]);**

**$I->assertEquals(1, $build->sequence);**

**$build = $I->have(Build::class, ['project\_id' => $this->project->id]);**

**$I->assertEquals(2, $build->sequence);**

**$this->project->refresh();**

**$I->assertEquals(3, $this->project->build\_sequence);**

**}**

This format can be recommended for testing domain and database interactions.

In Cest format you don’t have native support for test doubles so it’s recommended to include a trait \Codeception\Test\Feature\Stub to enable mocks inside a test. Alternatively, install and enable [Mockery module](https://github.com/Codeception/MockeryModule).

## ADVANCED TOOLS

### SPECIFY

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:

**class UserTest extends \Codeception\Test\Unit**

**{**

**use \Codeception\Specify;**

**/\*\* @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 = 'toolooooongnaaaaaaameeee';**

**$this->assertFalse($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 will not be reflected in other code blocks as it is marked with @specify annotation.

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:

**verify($user->getName())->equals('john');**

### DOMAIN ASSERTIONS

The more complicated your domain is the more explicit your tests should be. With [DomainAssert](https://github.com/Codeception/DomainAssert) library you can easily create custom assertion methods for unit and integration tests.

It allows to reuse business rules inside assertion methods:

**$user = new User;**

***// simple custom assertions below:***

**$this->assertUserIsValid($user);**

**$this->assertUserIsAdmin($user);**

***// use combined explicit assertion***

***// to tell what you expect to check***

**$this->assertUserCanPostToBlog($user, $blog);**

***// instead of just calling a bunch of assertions***

**$this->assertNotNull($user);**

**$this->assertNotNull($blog);**

**$this->assertContain($user, $blog->getOwners());**

With custom assertion methods you can improve readability of your tests and keep them focused around the specification.

### ASPECTMOCK

[AspectMock](https://github.com/Codeception/AspectMock) is an advanced mocking framework which allows you to replace any methods of any class in a test. Static methods, class methods, date and time functions can be easily replaced with AspectMock. For instance, you can test singletons!

**public function testSingleton()**

**{**

**$class = MySingleton::getInstance();**

**$this->assertInstanceOf('MySingleton', $class);**

**test::double('MySingleton', ['getInstance' => new DOMDocument]);**

**$this->assertInstanceOf('DOMDocument', $class);**

**}**

* [AspectMock on GitHub](https://github.com/Codeception/AspectMock)
* [AspectMock in Action](https://codeception.com/07-31-2013/nothing-is-untestable-aspect-mock.html)
* [How it Works](https://codeception.com/09-13-2013/understanding-aspectmock.html)

## 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 separately, 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 >**](https://codeception.com/docs/06-ModulesAndHelpers)
* **Previous Chapter:**[**< FunctionalTests**](https://codeception.com/docs/04-FunctionalTests)

# MODULES AND HELPERS

Codeception uses modularity to create a comfortable testing environment for every test suite you write.

All actions and assertions that can be performed by the Tester object in a class are defined in modules. You can extend the testing suite with your own actions and assertions by writing them into a custom module.

Let’s look at the following test:

**$I = new FunctionalTester($scenario);**

**$I->amOnPage('/');**

**$I->see('Hello');**

**$I->seeInDatabase('users', array('id' => 1));**

**$I->seeFileFound('running.lock');**

It can operate with different entities: the web page can be loaded with the PhpBrowser module, the database assertion uses the Db module, and file state can be checked with the Filesystem module.

Modules are attached to the Actor classes in the suite configuration. For example, in tests/functional.suite.yml we should see:

**actor: FunctionalTester**

**modules:**

**enabled:**

**- PhpBrowser:**

**url: http://localhost**

**- Db:**

**dsn: "mysql:host=localhost;dbname=testdb"**

**- Filesystem**

The FunctionalTester class has its methods defined in modules. Actually, it doesn’t contain any of them, but rather acts as a proxy. It knows which module executes this action and passes parameters into it. To make your IDE see all of the FunctionalTester methods, you should run the codecept build command. It generates method signatures from enabled modules and saves them into a trait which is included in an actor. In the current example, the tests/support/\_generated/FunctionalTesterActions.php file will be generated. By default, Codeception automatically rebuilds the Actions trait on each change of the suite configuration.

## STANDARD MODULES

Codeception has many bundled modules which will help you run tests for different purposes and different environments. The idea of modules is to share common actions, so that developers and QA engineers can concentrate on testing and not on reinventing the wheel. Each module provides methods for testing its own part and by combining modules you can get a powerful setup to test an application at all levels.

There is the WebDriver module for acceptance testing, modules for all popular PHP frameworks, PHPBrowser to emulate browser execution, REST for testing APIs, and more. Modules are considered to be the most valuable part of Codeception. They are constantly improving to provide the best testing experience, and be flexible to satisfy everyone’s needs.

### MODULE CONFLICTS

Modules may conflict with one another. If a module implements Codeception\Lib\Interfaces\ConflictsWithModule, it might declare a conflict rule to be used with other modules. For instance, WebDriver conflicts with all modules implementing the Codeception\Lib\Interfaces\Web interface.

**public function \_conflicts()**

**{**

**return 'Codeception\Lib\Interfaces\Web';**

**}**

This way if you try to use two modules sharing the same conflicted interface you will get an exception.

To avoid confusion, **Framework modules, PhpBrowser, and WebDriver** can’t be used together. For instance, the amOnPage method exists in all those modules, and you should not try to guess which module will actually execute it. If you are doing acceptance testing, set up either WebDriver or PHPBrowser but do not set both up at the same time. If you are doing functional testing, enable only one of the framework modules.

In case you need to use a module which depends on a conflicted one, specify it as a dependent module in the configuration. You may want to use WebDriver with the REST module which interacts with a server through PhpBrowser. In this case your config should look like this:

**modules:**

**enabled:**

**- WebDriver:**

**browser: firefox**

**url: http://localhost**

**- REST:**

**url: http://localhost/api/v1**

**depends: PhpBrowser**

This configuration will allow you to send GET/POST requests to the server’s APIs while working with a site through a browser.

If you only need some parts of a conflicted module to be loaded, refer to the next section.

### MODULE PARTS

Modules with Parts section in their reference can be partially loaded. This way, the $I object will have actions belonging to only a specific part of that module. Partially loaded modules can be also used to avoid module conflicts.

For instance, the Laravel5 module has an ORM part which contains database actions. You can enable the PhpBrowser module for testing and Laravel + ORM for connecting to the database and checking the data.

**modules:**

**enabled:**

**- PhpBrowser:**

**url: http://localhost**

**- Laravel5:**

**part: ORM**

The modules won’t conflict as actions with the same names won’t be loaded.

The REST module has parts for Xml and Json in the same way. If you are testing a REST service with only JSON responses, you can enable just the JSON part of this module:

**actor: ApiTester**

**modules:**

**enabled:**

**- REST:**

**url: http://serviceapp/api/v1/**

**depends: PhpBrowser**

**part: Json**

## HELPERS

Codeception doesn’t restrict you to only the modules from the main repository. Your project might need your own actions added to the test suite. By running the bootstrap command, Codeception generates three dummy modules for you, one for each of the newly created suites. These custom modules are called ‘Helpers’, and they can be found in the tests/\_support directory.

**namespace Helper;**

***// here you can define custom functions for FunctionalTester***

**class Functional extends \Codeception\Module**

**{**

**}**

Actions are also quite simple. Every action you define is a public function. Write a public method, then run the build command, and you will see the new function added into the FunctionalTester class.

Public methods prefixed by `\_` are treated as hidden and won't be added to your Actor class.

Assertions can be a bit tricky. First of all, it’s recommended to prefix all your assertion actions with see or dontSee.

Name your assertions like this:

**$I->seePageReloaded();**

**$I->seeClassIsLoaded($classname);**

**$I->dontSeeUserExist($user);**

And then use them in your tests:

**$I->seePageReloaded();**

**$I->seeClassIsLoaded('FunctionalTester');**

**$I->dontSeeUserExist($user);**

You can define assertions by using assertXXX methods in your modules.

**function seeClassExist($class)**

**{**

**$this->assertTrue(class\_exists($class));**

**}**

In your helpers you can use these assertions:

**function seeCanCheckEverything($thing)**

**{**

**$this->assertTrue(isset($thing), "this thing is set");**

**$this->assertFalse(empty($any), "this thing is not empty");**

**$this->assertNotNull($thing, "this thing is not null");**

**$this->assertContains("world", $thing, "this thing contains 'world'");**

**$this->assertNotContains("bye", $thing, "this thing doesn't contain 'bye'");**

**$this->assertEquals("hello world", $thing, "this thing is 'Hello world'!");**

***// ...***

**}**

### ACCESSING OTHER MODULES

It’s possible that you will need to access internal data or functions from other modules. For example, for your module you might need to access the responses or internal actions of other modules.

Modules can interact with each other through the getModule method. Please note that this method will throw an exception if the required module was not loaded.

Let’s imagine that we are writing a module that reconnects to a database. It’s supposed to use the dbh connection value from the Db module.

**function reconnectToDatabase()**

**{**

**$dbh = $this->getModule('Db')->dbh;**

**$dbh->close();**

**$dbh->open();**

**}**

By using the getModule function, you get access to all of the public methods and properties of the requested module. The dbh property was defined as public specifically to be available to other modules.

Modules may also contain methods that are exposed for use in helper classes. Those methods start with a \_ prefix and are not available in Actor classes, so can be accessed only from modules and extensions.

You should use them to write your own actions using module internals.

**function seeNumResults($num)**

**{**

***// retrieving webdriver session***

**/\*\*@var $table \Facebook\WebDriver\WebDriverElement \*/**

**$elements = $this->getModule('WebDriver')->\_findElements('#result');**

**$this->assertNotEmpty($elements);**

**$table = reset($elements);**

**$this->assertEquals('table', $table->getTagName());**

**$results = $table->findElements('tr');**

***// asserting that table contains exactly $num rows***

**$this->assertEquals($num, count($results));**

**}**

In this example we use the API of the [facebook/php-webdriver](https://github.com/facebook/php-webdriver) library, a Selenium WebDriver client the module is build on. You can also access the webDriver property of a module to get access to the Facebook\WebDriver\RemoteWebDriver instance for direct Selenium interaction.

### EXTENDING A MODULE

If accessing modules doesn’t provide enough flexibility, you can extend a module inside a Helper class:

**namespace Helper;**

**class MyExtendedSelenium extends \Codeception\Module\WebDriver**

**{**

**}**

In this helper you can replace the parent’s methods with your own implementation. You can also replace the \_before and \_after hooks, which might be an option when you need to customize starting and stopping of a testing session.

### HOOKS

Each module can handle events from the running test. A module can be executed before the test starts, or after the test is finished. This can be useful for bootstrap/cleanup actions. You can also define special behavior for when the test fails. This may help you in debugging the issue. For example, the PhpBrowser module saves the current webpage to the tests/\_output directory when a test fails.

All hooks are defined in [Codeception\Module](https://codeception.com/docs/reference/Commands) and are listed here. You are free to redefine them in your module.

***// HOOK: used after configuration is loaded***

**public function \_initialize()**

**{**

**}**

***// HOOK: before each suite***

**public function \_beforeSuite($settings = array())**

**{**

**}**

***// HOOK: after suite***

**public function \_afterSuite()**

**{**

**}**

***// HOOK: before each step***

**public function \_beforeStep(\Codeception\Step $step)**

**{**

**}**

***// HOOK: after each step***

**public function \_afterStep(\Codeception\Step $step)**

**{**

**}**

***// HOOK: before test***

**public function \_before(\Codeception\TestInterface $test)**

**{**

**}**

***// HOOK: after test***

**public function \_after(\Codeception\TestInterface $test)**

**{**

**}**

***// HOOK: on fail***

**public function \_failed(\Codeception\TestInterface $test, $fail)**

**{**

**}**

Please note that methods with a \_ prefix are not added to the Actor class. This allows them to be defined as public but used only for internal purposes.

### DEBUG

As we mentioned, the \_failed hook can help in debugging a failed test. You have the opportunity to save the current test’s state and show it to the user, but you are not limited to this.

Each module can output internal values that may be useful during debug. For example, the PhpBrowser module prints the response code and current URL every time it moves to a new page. Thus, modules are not black boxes. They are trying to show you what is happening during the test. This makes debugging your tests less painful.

To display additional information, use the debug and debugSection methods of the module. Here is an example of how it works for PhpBrowser:

**$this->debugSection('Request', $params);**

**$this->client->request($method, $uri, $params);**

**$this->debug('Response Code: ' . $this->client->getStatusCode());**

This test, running with the PhpBrowser module in debug mode, will print something like this:

**I click "All pages"**

**\* Request (GET) http://localhost/pages {}**

**\* Response code: 200**

## CONFIGURATION

Modules and Helpers can be configured from the suite configuration file, or globally from codeception.yml.

Mandatory parameters should be defined in the $requiredFields property of the class. Here is how it is done in the Db module:

**class Db extends \Codeception\Module**

**{**

**protected $requiredFields = ['dsn', 'user', 'password'];**

***// ...***

**}**

The next time you start the suite without setting one of these values, an exception will be thrown.

For optional parameters, you should set default values. The $config property is used to define optional parameters as well as their values. In the WebDriver module we use the default Selenium Server address and port.

**class WebDriver extends \Codeception\Module**

**{**

**protected $requiredFields = ['browser', 'url'];**

**protected $config = ['host' => '127.0.0.1', 'port' => '4444'];**

***// ...***

**}**

The host and port parameter can be redefined in the suite configuration. Values are set in the modules:config section of the configuration file.

**modules:**

**enabled:**

**- WebDriver:**

**url: 'http://mysite.com/'**

**browser: 'firefox'**

**- Db:**

**cleanup: false**

**repopulate: false**

Optional and mandatory parameters can be accessed through the $config property. Use $this->config['parameter'] to get its value.

### DYNAMIC CONFIGURATION WITH PARAMETERS

Modules can be dynamically configured from environment variables. Parameter storage should be specified in the global codeception.yml configuration inside the params section. Parameters can be loaded from environment vars, from yaml (Symfony format), .env (Laravel format), ini, or php files.

Use the params section of the global configuration file codeception.yml to specify how to load them. You can specify several sources for parameters to be loaded from.

Example: load parameters from the environment:

**params:**

**- env *# load params from environment vars***

Example: load parameters from YAML file (Symfony):

**params:**

**- app/config/parameters.yml**

Example: load parameters from php file (Yii)

**params:**

**- config/params.php**

Example: load parameters from .env files (Laravel):

**params:**

**- .env**

**- .env.testing**

Once loaded, parameter variables can be used as module configuration values. Use a variable name wrapped with % as a placeholder and it will be replaced by its value.

Let’s say we want to specify credentials for a cloud testing service. We have loaded SAUCE\_USER and SAUCE\_KEY variables from environment, and now we are passing their values into config of WebDriver:

**modules:**

**enabled:**

**- WebDriver:**

**url: http://mysite.com**

**host: '%SAUCE\_USER%:%SAUCE\_KEY%@ondemand.saucelabs.com'**

Parameters are also useful to provide connection credentials for the Db module (taken from Laravel’s .env files):

**modules:**

**enabled:**

**- Db:**

**dsn: "mysql:host=%DB\_HOST%;dbname=%DB\_DATABASE%"**

**user: "%DB\_USERNAME%"**

**password: "%DB\_PASSWORD%"**

### RUNTIME CONFIGURATION

If you want to reconfigure a module at runtime, you need to call a [helper](https://codeception.com/docs/06-ModulesAndHelpers#Helpers) that uses the \_reconfigure method of the module.

In this example we change the root URL for PhpBrowser, so that amOnPage('/') will open /admin/.

**$this->getModule('PhpBrowser')->\_reconfigure(['url' => 'http://localhost/admin']);**

Usually, these configuration changes are effective immediately. However, in WebDriver configuration changes can’t be applied that easily. For instance, if you change the browser you need to close the current browser session and start a new one. For that, WebDriver module provides a \_restart method which takes a config array and restarts the browser:

***// start chrome***

**$this->getModule('WebDriver')->\_restart(['browser' => 'chrome']);**

***// or just restart browser***

**$this->getModule('WebDriver')->\_restart();**

At the end of a test all configuration changes will be rolled back to the original configuration values.

### RUNTIME CONFIGURATION OF A TEST

Sometimes it is needed to set custom configuration for a specific test only. For [Cest](https://codeception.com/docs/07-AdvancedUsage#Cest-Classes) and [Test\Unit](https://codeception.com/docs/05-UnitTests) formats you can use @prepare annotation which can execute the code before other hooks are executed. This allows @prepare to change the module configuration in runtime. @prepare uses [dependency injection](https://codeception.com/docs/07-AdvancedUsage#Dependency-Injection) to automatically inject required modules into a method.

To run a specific test only in Chrome browser, you can call \_reconfigure from WebDriver module for a test itself using @prepare.

**/\*\***

**\* @prepare useChrome**

**\*/**

**public function chromeSpecificTest()**

**{**

***// ...***

**}**

**protected function useChrome(\Codeception\Module\WebDriver $webdriver)**

**{**

***// WebDriver was injected by the class name***

**$webdriver->\_reconfigure(['browser' => 'chrome']);**

**}**

Prepare methods can invoke all methods of a module, as well as hidden API methods (starting with \_). Use them to customize the module setup for a specific test.

To change module configuration for a specific group of tests use [GroupObjects](https://codeception.com/docs/08-Customization#Group-Objects).

## CONCLUSION

Modules are the real power of Codeception. They are used to emulate multiple inheritances for Actor classes (UnitTester, FunctionalTester, AcceptanceTester, etc). Codeception provides modules to emulate web requests, access data, interact with popular PHP libraries, etc. If the bundled modules are not enough for you that’s OK, you are free to write your own! Use Helpers (custom modules) for everything that Codeception can’t do out of the box. Helpers also can be used to extend the functionality of the original modules.

* **Next Chapter:**[**ReusingTestCode >**](https://codeception.com/docs/06-ReusingTestCode)
* **Previous Chapter:**[**< UnitTests**](https://codeception.com/docs/05-UnitTests)

**REUSING TEST CODE**

Codeception uses modularity to create a comfortable testing environment for every test suite you write. Modules allow you to choose the actions and assertions that can be performed in tests.

WHAT ARE ACTORS

All actions and assertions that can be performed by the Actor object in a class are defined in modules. It might look like Codeception limits you in testing, but that’s not true. You can extend the testing suite with your own actions and assertions, by writing them into a custom module, called a Helper. We will get back to this later in this chapter, but for now let’s look at the following test:

**$I->amOnPage('/');**

**$I->see('Hello');**

**$I->seeInDatabase('users', ['id' => 1]);**

**$I->seeFileFound('running.lock');**

It can operate with different entities: the web page can be loaded with the PhpBrowser module, the database assertion uses the Db module, and the file state can be checked with the Filesystem module.

Modules are attached to Actor classes in the suite config. For example, in tests/acceptance.suite.yml we should see:

**actor: AcceptanceTester**

**modules:**

**enabled:**

**- PhpBrowser:**

**url: http://localhost**

**- Db**

**- Filesystem**

The AcceptanceTester class has its methods defined in modules. Let’s see what’s inside the AcceptanceTester class, which is located inside the tests/\_support directory:

**/\*\***

**\* Inherited Methods**

**\* @method void wantToTest($text)**

**\* @method void wantTo($text)**

**\* @method void execute($callable)**

**\* @method void expectTo($prediction)**

**\* @method void expect($prediction)**

**\* @method void amGoingTo($argumentation)**

**\* @method void am($role)**

**\* @method void lookForwardTo($achieveValue)**

**\* @method void comment($description)**

**\* @method void haveFriend($name, $actorClass = null)**

**\***

**\* @SuppressWarnings(PHPMD)**

**\*/**

**class AcceptanceTester extends \Codeception\Actor**

**{**

**use \_generated\AcceptanceTesterActions;**

**/\*\***

**\* Define custom actions here**

**\*/**

**}**

The most important part is the \_generated\AcceptanceTesterActions trait, which is used as a proxy for enabled modules. It knows which module executes which action and passes parameters into it. This trait was created by running codecept build and is regenerated each time module or configuration changes.

Use actor classes to set common actions which can be used accross a suite.

**PAGEOBJECTS**

For acceptance and functional testing, we will not only need to have common actions being reused across different tests, we should have buttons, links and form fields being reused as well. For those cases we need to implement the [PageObject pattern](https://www.selenium.dev/documentation/en/guidelines_and_recommendations/page_object_models/), which is widely used by test automation engineers. The PageObject pattern represents a web page as a class and the DOM elements on that page as its properties, and some basic interactions as its methods. PageObjects are very important when you are developing a flexible architecture of your acceptance or functional tests. Do not hard-code complex CSS or XPath locators in your tests but rather move them into PageObject classes.

Codeception can generate a PageObject class for you with command:

**php vendor/bin/codecept generate:pageobject acceptance Login**

It is recommended to use page objects for acceptance testing only

This will create a Login class in tests/\_support/Page/Acceptance. The basic PageObject is nothing more than an empty class with a few stubs.

It is expected that you will populate it with the UI locators of a page it represents. Locators can be added as public properties:

**namespace Page\Acceptance;**

**class Login**

**{**

**public static $URL = '/login';**

**public $usernameField = '#mainForm #username';**

**public $passwordField = '#mainForm input[name=password]';**

**public $loginButton = '#mainForm input[type=submit]';**

***// ...***

**}**

But let’s move further. The PageObject concept specifies that the methods for the page interaction should also be stored in a PageObject class.

Let’s define a login method in this class:

**namespace Page\Acceptance;**

**class Login**

**{**

**public static $URL = '/login';**

**public $usernameField = '#mainForm #username';**

**public $passwordField = '#mainForm input[name=password]';**

**public $loginButton = '#mainForm input[type=submit]';**

**/\*\***

**\* @var AcceptanceTester**

**\*/**

**protected $tester;**

***// we inject AcceptanceTester into our class***

**public function \_\_construct(\AcceptanceTester $I)**

**{**

**$this->tester = $I;**

**}**

**public function login($name, $password)**

**{**

**$I = $this->tester;**

**$I->amOnPage(self::$URL);**

**$I->fillField($this->usernameField, $name);**

**$I->fillField($this->passwordField, $password);**

**$I->click($this->loginButton);**

**}**

**}**

If you specify which object you need for a test, Codeception will try to create it using the dependency injection container. In the case of a PageObject you should declare a class as a parameter for a test method:

**class UserCest**

**{**

**function showUserProfile(AcceptanceTester $I, \Page\Acceptance\Login $loginPage)**

**{**

**$loginPage->login('bill evans', 'debby');**

**$I->amOnPage('/profile');**

**$I->see('Bill Evans Profile', 'h1');**

**}**

**}**

The dependency injection container can construct any object that requires any known class type. For instance, Page\Login required AcceptanceTester, and so it was injected into Page\Login constructor, and PageObject was created and passed into method arguments. You should explicitly specify the types of required objects for Codeception to know what objects should be created for a test. Dependency Injection will be described in the next chapter.

**STEPOBJECTS**

StepObjects are great if you need some common functionality for a group of tests. Let’s say you are going to test an admin area of a site. You probably won’t need the same actions from the admin area while testing the front end, so it’s a good idea to move these admin-specific tests into their own class. We call such a classes StepObjects.

Lets create an Admin StepObject with the generator:

**php vendor/bin/codecept generate:stepobject acceptance Admin**

You can supply optional action names. Enter one at a time, followed by a newline. End with an empty line to continue to StepObject creation.

**php vendor/bin/codecept generate:stepobject acceptance Admin**

**Add action to StepObject class (ENTER to exit): loginAsAdmin**

**Add action to StepObject class (ENTER to exit):**

**StepObject was created in /tests/acceptance/\_support/Step/Acceptance/Admin.php**

This will generate a class in /tests/\_support/Step/Acceptance/Admin.php similar to this:

**namespace Step\Acceptance;**

**class Admin extends \AcceptanceTester**

**{**

**public function loginAsAdmin()**

**{**

**$I = $this;**

**}**

**}**

As you see, this class is very simple. It extends the AcceptanceTester class, meaning it can access all the methods and properties of AcceptanceTester.

The loginAsAdmin method may be implemented like this:

**namespace Step\Acceptance;**

**class Admin extends \AcceptanceTester**

**{**

**public function loginAsAdmin()**

**{**

**$I = $this;**

**$I->amOnPage('/admin');**

**$I->fillField('username', 'admin');**

**$I->fillField('password', '123456');**

**$I->click('Login');**

**}**

**}**

StepObject can be instantiated automatically when used inside the Cest format:

**class UserCest**

**{**

**function showUserProfile(\Step\Acceptance\Admin $I)**

**{**

**$I->loginAsAdmin();**

**$I->amOnPage('/admin/profile');**

**$I->see('Admin Profile', 'h1');**

**}**

**}**

If you have a complex interaction scenario, you may use several step objects in one test. If you feel like adding too many actions into your Actor class (which is AcceptanceTester in this case) consider moving some of them into separate StepObjects.

Use StepObjects when you have multiple areas of applications or multiple roles.

**CONCLUSION**

There are lots of ways to create reusable and readable tests. Group common actions together and move them to an Actor class or StepObjects. Move CSS and XPath locators into PageObjects. Write your custom actions and assertions in Helpers. Scenario-driven tests should not contain anything more complex than $I->doSomething commands. Following this approach will allow you to keep your tests clean, readable, stable and make them easy to maintain.

* **Next Chapter:**[**AdvancedUsage >**](https://codeception.com/docs/07-AdvancedUsage)
* **Previous Chapter:**[**< ModulesAndHelpers**](https://codeception.com/docs/06-ModulesAndHelpers)

Don't know how to write tests on your own? We will build or improve them for you. **Request a quote for official**[**enterprise support**](http://sdclabs.com/codeception?utm_source=codeception.com&utm_medium=docs_bottom&utm_term=link&utm_campaign=reference)**or**[**trainings**](http://sdclabs.com/trainings?utm_source=codeception.com&utm_medium=docs_bottom&utm_term=link&utm_campaign=reference)

# ADVANCED USAGE

In this chapter we will cover some techniques and options that you can use to improve your testing experience and keep your project better organized.

## CEST CLASSES

If you want to get a class-like structure for your Cepts, you can use the Cest format instead of plain PHP. It is very simple and is fully compatible with Cept scenarios. It means that if you feel that your test is long enough and you want to split it, you can easily move it into classes.

You can create a Cest file by running the command:

**php vendor/bin/codecept generate:cest suitename CestName**

The generated file will look like this:

**class BasicCest**

**{**

**public function \_before(\AcceptanceTester $I)**

**{**

**}**

**public function \_after(\AcceptanceTester $I)**

**{**

**}**

***// tests***

**public function tryToTest(\AcceptanceTester $I)**

**{**

**}**

**}**

**Each public method of Cest (except those starting with \_) will be executed as a test** and will receive an instance of the Actor class as the first parameter and the $scenario variable as the second one.

In \_before and \_after methods you can use common setups and teardowns for the tests in the class.

As you see, we are passing the Actor object into tryToTest method. This allows us to write scenarios the way we did before:

**class BasicCest**

**{**

***// test***

**public function tryToTest(\AcceptanceTester $I)**

**{**

**$I->amOnPage('/');**

**$I->click('Login');**

**$I->fillField('username', 'john');**

**$I->fillField('password', 'coltrane');**

**$I->click('Enter');**

**$I->see('Hello, John');**

**$I->seeInCurrentUrl('/account');**

**}**

**}**

As you see, Cest classes have no parents. This is done intentionally. It allows you to extend your classes with common behaviors and workarounds that may be used in child classes. But don’t forget to make these methods protected so they won’t be executed as tests.

Cest format also can contain hooks based on test results:

* \_failed will be executed on failed test
* \_passed will be executed on passed test

**public function \_failed(\AcceptanceTester $I)**

**{**

***// will be executed on test failure***

**}**

**public function \_passed(\AcceptanceTester $I)**

**{**

***// will be executed when test is successful***

**}**

## DEPENDENCY INJECTION

Codeception supports simple dependency injection for Cest and \Codeception\TestCase\Test classes. It means that you can specify which classes you need as parameters of the special \_inject() method, and Codeception will automatically create the respective objects and invoke this method, passing all dependencies as arguments. This may be useful when working with Helpers. Here’s an example for Cest:

**class SignUpCest**

**{**

**/\*\***

**\* @var Helper\SignUp**

**\*/**

**protected $signUp;**

**/\*\***

**\* @var Helper\NavBarHelper**

**\*/**

**protected $navBar;**

**protected function \_inject(\Helper\SignUp $signUp, \Helper\NavBar $navBar)**

**{**

**$this->signUp = $signUp;**

**$this->navBar = $navBar;**

**}**

**public function signUp(\AcceptanceTester $I)**

**{**

**$this->navBar->click('Sign up');**

**$this->signUp->register([**

**'first\_name' => 'Joe',**

**'last\_name' => 'Jones',**

**'email' => 'joe@jones.com',**

**'password' => '1234',**

**'password\_confirmation' => '1234'**

**]);**

**}**

**}**

And for Test classes:

**class MathTest extends \Codeception\TestCase\Test**

**{**

**/\*\***

**\* @var \UnitTester**

**\*/**

**protected $tester;**

**/\*\***

**\* @var Helper\Math**

**\*/**

**protected $math;**

**protected function \_inject(\Helper\Math $math)**

**{**

**$this->math = $math;**

**}**

**public function testAll()**

**{**

**$this->assertEquals(3, $this->math->add(1, 2));**

**$this->assertEquals(1, $this->math->subtract(3, 2));**

**}**

**}**

However, Dependency Injection is not limited to this. It allows you to **inject any class**, which can be constructed with arguments known to Codeception.

In order to make auto-wiring work, you will need to implement the \_inject() method with the list of desired arguments. It is important to specify the type of arguments, so Codeception can guess which objects are expected to be received. The \_inject() will only be invoked once, just after creation of the TestCase object (either Cest or Test). Dependency Injection will also work in a similar manner for Helper and Actor classes.

Each test of a Cest class can declare its own dependencies and receive them from method arguments:

**class UserCest**

**{**

**function updateUser(\Helper\User $u, \AcceptanceTester $I, \Page\User $userPage)**

**{**

**$user = $u->createDummyUser();**

**$userPage->login($user->getName(), $user->getPassword());**

**$userPage->updateProfile(['name' => 'Bill']);**

**$I->see('Profile was saved');**

**$I->see('Profile of Bill','h1');**

**}**

**}**

Moreover, Codeception can resolve dependencies recursively (when A depends on B, and B depends on C etc.) and handle parameters of primitive types with default values (like $param = 'default'). Of course, you are not allowed to have cyclic dependencies.

## EXAMPLE ANNOTATION

What if you want to execute the same test scenario with different data? In this case you can inject examples as \Codeception\Example instances. Data is defined via the @example annotation, using JSON or Doctrine-style notation (limited to a single line). Doctrine-style:

**class EndpointCest**

**{**

**/\*\***

**\* @example ["/api/", 200]**

**\* @example ["/api/protected", 401]**

**\* @example ["/api/not-found-url", 404]**

**\* @example ["/api/faulty", 500]**

**\*/**

**public function checkEndpoints(ApiTester $I, \Codeception\Example $example)**

**{**

**$I->sendGet($example[0]);**

**$I->seeResponseCodeIs($example[1]);**

**}**

**}**

JSON:

**class PageCest**

**{**

**/\*\***

**\* @example { "url": "/", "title": "Welcome" }**

**\* @example { "url": "/info", "title": "Info" }**

**\* @example { "url": "/about", "title": "About Us" }**

**\* @example { "url": "/contact", "title": "Contact Us" }**

**\*/**

**public function staticPages(AcceptanceTester $I, \Codeception\Example $example)**

**{**

**$I->amOnPage($example['url']);**

**$I->see($example['title'], 'h1');**

**$I->seeInTitle($example['title']);**

**}**

**}**

If you use JSON notation please keep in mind that all string keys and values should be enclosed in double quotes (`"`) according to JSON standard.

Key-value data in Doctrine-style annotation syntax:

**class PageCest**

**{**

**/\*\***

**\* @example(url="/", title="Welcome")**

**\* @example(url="/info", title="Info")**

**\* @example(url="/about", title="About Us")**

**\* @example(url="/contact", title="Contact Us")**

**\*/**

**public function staticPages(AcceptanceTester $I, \Codeception\Example $example)**

**{**

**$I->amOnPage($example['url']);**

**$I->see($example['title'], 'h1');**

**$I->seeInTitle($example['title']);**

**}**

**}**

## DATAPROVIDER ANNOTATIONS

You can also use the @dataProvider annotation for creating dynamic examples for [Cest classes](https://codeception.com/docs/07-AdvancedUsage#Cest-Classes), using a **protected method** for providing example data:

**class PageCest**

**{**

**/\*\***

**\* @dataProvider pageProvider**

**\*/**

**public function staticPages(AcceptanceTester $I, \Codeception\Example $example)**

**{**

**$I->amOnPage($example['url']);**

**$I->see($example['title'], 'h1');**

**$I->seeInTitle($example['title']);**

**}**

**/\*\***

**\* @return array**

**\*/**

**protected function pageProvider() *// alternatively, if you want the function to be public, be sure to prefix it with `\_`***

**{**

**return [**

**['url'=>"/", 'title'=>"Welcome"],**

**['url'=>"/info", 'title'=>"Info"],**

**['url'=>"/about", 'title'=>"About Us"],**

**['url'=>"/contact", 'title'=>"Contact Us"]**

**];**

**}**

**}**

@dataprovider annotation is also available for [unit tests](https://codeception.com/docs/05-UnitTests), in this case the data provider **method must be public**. For more details about how to use data provider for unit tests, please refer to [PHPUnit documentation](https://phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.data-providers).

## BEFORE/AFTER ANNOTATIONS

You can control execution flow with @before and @after annotations. You may move common actions into protected (non-test) methods and invoke them before or after the test method by putting them into annotations. It is possible to invoke several methods by using more than one @before or @after annotation. Methods are invoked in order from top to bottom.

**class ModeratorCest {**

**protected function login(AcceptanceTester $I)**

**{**

**$I->amOnPage('/login');**

**$I->fillField('Username', 'miles');**

**$I->fillField('Password', 'davis');**

**$I->click('Login');**

**}**

**/\*\***

**\* @before login**

**\*/**

**public function banUser(AcceptanceTester $I)**

**{**

**$I->amOnPage('/users/charlie-parker');**

**$I->see('Ban', '.button');**

**$I->click('Ban');**

**}**

**/\*\***

**\* @before login**

**\* @before cleanup**

**\* @after logout**

**\* @after close**

**\*/**

**public function addUser(AcceptanceTester $I)**

**{**

**$I->amOnPage('/users/charlie-parker');**

**$I->see('Ban', '.button');**

**$I->click('Ban');**

**}**

**}**

## ENVIRONMENTS

For cases where you need to run tests with different configurations you can define different config environments. The most typical use cases are running acceptance tests in different browsers, or running database tests using different database engines.

Let’s demonstrate the usage of environments for the browsers case.

We need to add some new lines to acceptance.suite.yml:

**actor: AcceptanceTester**

**modules:**

**enabled:**

**- WebDriver**

**- \Helper\Acceptance**

**config:**

**WebDriver:**

**url: 'http://127.0.0.1:8000/'**

**browser: 'firefox'**

**chrome:**

**modules:**

**config:**

**WebDriver:**

**browser: 'chrome'**

**firefox:**

***# nothing changed***

Basically you can define different environments inside the env root, name them (chrome, firefox etc.), and then redefine any configuration parameters that were set before.

You can also define environments in separate configuration files placed in the directory specified by the envs option in the paths configuration:

**paths:**

**envs: tests/\_envs**

The names of these files are used as environments names (e.g. chrome.yml or chrome.dist.yml for an environment named chrome). You can generate a new file with this environment configuration by using the generate:environment command:

**$ php vendor/bin/codecept g:env chrome**

In that file you can specify just the options you wish to override:

**modules:**

**config:**

**WebDriver:**

**browser: 'chrome'**

The environment configuration files are merged into the main configuration before the suite configuration is merged.

You can easily switch between those configs by running tests with --env option. To run the tests only for Firefox you just need to pass --env firefox as an option:

**$ php vendor/bin/codecept run acceptance --env firefox**

To run the tests in all browsers, list all the environments:

**$ php vendor/bin/codecept run acceptance --env chrome --env firefox**

The tests will be executed 3 times, each time in a different browser.

It’s also possible to merge multiple environments into a single configuration by separating them with a comma:

**$ php vendor/bin/codecept run acceptance --env dev,firefox --env dev,chrome --env dev,firefox**

The configuration is merged in the order given. This way you can easily create multiple combinations of your environment configurations.

Depending on the environment, you may choose which tests are to be executed. For example, you might need some tests to be executed in Firefox only, and some tests in Chrome only.

The desired environments can be specified with the @env annotation for tests in Test and Cest formats:

**class UserCest**

**{**

**/\*\***

**\* This test will be executed only in 'firefox' and 'chrome' environments**

**\***

**\* @env firefox**

**\* @env chrome**

**\*/**

**public function webkitOnlyTest(AcceptanceTester $I)**

**{**

***// I do something***

**}**

**}**

For Cept you should use simple comments:

***// @env firefox***

***// @env chrome***

This way you can easily control which tests will be executed for each environment.

## GET SCENARIO METADATA

Sometimes you may need to change the test behavior in real time. For instance, the behavior of the same test may differ in Firefox and in Chrome. In runtime we can retrieve the current environment name, test name, or list of enabled modules by calling the $scenario->current() method.

***// retrieve current environment***

**$scenario->current('env');**

***// list of all enabled modules***

**$scenario->current('modules');**

***// test name***

**$scenario->current('name');**

***// browser name (if WebDriver module enabled)***

**$scenario->current('browser');**

***// capabilities (if WebDriver module enabled)***

**$scenario->current('capabilities');**

You can access \Codeception\Scenario in the Cept and Cest formats. In Cept, the $scenario variable is available by default, while in Cest you should retrieve it through dependency injection:

**public function myTest(\AcceptanceTester $I, \Codeception\Scenario $scenario)**

**{**

**if ($scenario->current('browser') == 'chrome') {**

***// ...***

**}**

**}**

Codeception\Scenario is also available in Actor classes and StepObjects. You can access it with $this->getScenario().

## SHUFFLE

By default Codeception runs tests in alphabetic order. To ensure that tests are not depending on each other (unless explicitly declared via @depends) you can enable shuffle option.

***# inside codeception.yml***

**settings:**

**shuffle: true**

Alternatively, you may run tests in shuffle without changing the config:

**codecept run -o "settings: shuffle: true"**

Tests will be randomly reordered on each run. When tests executed in shuffle mode a seed value will be printed. Copy this seed value from output to be able to rerun tests in the same order.

**$ codecept run**

**Codeception PHP Testing Framework v2.4.5**

**Powered by PHPUnit 5.7.27 by Sebastian Bergmann and contributors.**

**[Seed] 1872290562**

Pass the copied seed into --seed option:

**codecept run --seed 1872290562**

### DEPENDENCIES

With the @depends annotation you can specify a test that should be passed before the current one. If that test fails, the current test will be skipped. You should pass the method name of the test you are relying on.

**class ModeratorCest {**

**public function login(AcceptanceTester $I)**

**{**

***// logs moderator in***

**}**

**/\*\***

**\* @depends login**

**\*/**

**public function banUser(AcceptanceTester $I)**

**{**

***// bans user***

**}**

**}**

@depends applies to the Cest and Codeception\Test\Unit formats. Dependencies can be set across different classes. To specify a dependent test from another file you should provide a test signature. Normally, the test signature matches the className:methodName format. But to get the exact test signature just run the test with the --steps option to see it:

**Signature: ModeratorCest:login`**

Codeception reorders tests so dependent tests will always be executed before the tests that rely on them.

## RUNNING FROM DIFFERENT FOLDERS

If you have several projects with Codeception tests, you can use a single codecept file to run all of your tests. You can pass the -c option to any Codeception command (except bootstrap), to execute Codeception in another directory:

**$ php vendor/bin/codecept run -c ~/projects/ecommerce/**

**$ php vendor/bin/codecept run -c ~/projects/drupal/**

**$ php vendor/bin/codecept generate:cept acceptance CreateArticle -c ~/projects/drupal/**

To create a project in directory different from the current one, just provide its path as a parameter:

**$ php vendor/bin/codecept bootstrap ~/projects/drupal/**

Also, the -c option allows you to specify another config file to be used. Thus, you can have several codeception.yml files for your test suite (e.g. to to specify different environments and settings). Just pass the .yml filename as the -c parameter to execute tests with specific config settings.

## GROUPS

There are several ways to execute a bunch of tests. You can run tests from a specific directory:

**$ php vendor/bin/codecept run tests/acceptance/admin**

You can execute one (or several) specific groups of tests:

**$ php vendor/bin/codecept run -g admin -g editor**

The concept of groups was taken from PHPUnit and behave in the same way.

For Test and Cest files you can use the @group annotation to add a test to a group.

**/\*\***

**\* @group admin**

**\*/**

**public function testAdminUser()**

**{**

**}**

For Cept files, use pseudo-annotations in comments:

***// @group admin***

***// @group editor***

**$I = new AcceptanceTester($scenario);**

**$I->wantToTest('admin area');**

For .feature-files (Gherkin) use tags:

**@admin @editor**

**Feature: Admin area**

### GROUP FILES

Groups can be defined in global or suite configuration files. Tests for groups can be specified as an array of file names or directories containing them:

**groups:**

***# add 2 tests to db group***

**db: [tests/unit/PersistTest.php, tests/unit/DataTest.php]**

***# add all tests from a directory to api group***

**api: [tests/functional/api]**

A list of tests for the group can be passed from a Group file. It should be defined in plain text with test names on separate lines:

**tests/unit/DbTest.php**

**tests/unit/UserTest.php:create**

**tests/unit/UserTest.php:update**

A group file can be included by its relative filename:

**groups:**

***# requiring a group file***

**slow: tests/\_data/slow.txt**

You can create group files manually or generate them from third party applications. For example, you can write a script that updates the slow group by taking the slowest tests from xml report.

You can even specify patterns for loading multiple group files with a single definition:

**groups:**

**p\*: tests/\_data/p\***

This will load all found p\* files in tests/\_data as groups. Group names will be as follows p1,p2,…,pN.

## FORMATS

In addition to the standard test formats (Cept, Cest, Unit, Gherkin) you can implement your own format classes to customise your test execution. Specify these in your suite configuration:

**formats:**

**- \My\Namespace\MyFormat**

Then define a class which implements the LoaderInterface

**namespace My\Namespace;**

**class MyFormat implements \Codeception\Test\Loader\LoaderInterface**

**{**

**protected $tests;**

**protected $settings;**

**public function \_\_construct($settings = [])**

**{**

***//These are the suite settings***

**$this->settings = $settings;**

**}**

**public function loadTests($filename)**

**{**

***//Load file and create tests***

**}**

**public function getTests()**

**{**

**return $this->tests;**

**}**

**public function getPattern()**

**{**

**return '~Myformat\.php$~';**

**}**

**}**

## SHELL AUTO-COMPLETION

For bash and zsh shells, you can use auto-completion for your Codeception projects by executing the following in your shell (or add it to your .bashrc/.zshrc):

***# BASH ~4.x, ZSH***

**source <([codecept location] \_completion --generate-hook --program codecept --use-vendor-bin)**

***# BASH ~3.x, ZSH***

**[codecept location] \_completion --generate-hook --program codecept --use-vendor-bin | source /dev/stdin**

***# BASH (any version)***

**eval *$(*[codecept location] \_completion --generate-hook --program codecept --use-vendor-bin*)***

### EXPLANATION

By using the above code in your shell, Codeception will try to autocomplete the following:

* Commands
* Suites
* Test paths

Usage of -use-vendor-bin is optional. This option will work for most Codeception projects, where Codeception is located in your vendor/bin folder. But in case you are using a global Codeception installation for example, you wouldn’t use this option.

Note that with the -use-vendor-bin option, your commands will be completed using the Codeception binary located in your project’s root. Without the option, it will use whatever Codeception binary you originally used to generate the completion script (‘codecept location’ in the above examples)

## CONCLUSION

Codeception is a framework which may look simple at first glance but it allows you to build powerful tests with a single API, refactor them, and write them faster using the interactive console. Codeception tests can be easily organized in groups or Cest classes.

* **Next Chapter:**[**BDD >**](https://codeception.com/docs/07-BDD)
* **Previous Chapter:**[**< ReusingTestCode**](https://codeception.com/docs/06-ReusingTestCode)

# BEHAVIOR DRIVEN DEVELOPMENT

Behavior Driven Development (BDD) is a popular software development methodology. BDD is considered an extension of TDD, and is greatly inspired by [Agile](http://agilemanifesto.org/) practices. The primary reason to choose BDD as your development process is to break down communication barriers between business and technical teams. BDD encourages the use of automated testing to verify all documented features of a project from the very beginning. This is why it is common to talk about BDD in the context of test frameworks (like Codeception). The BDD approach, however, is about much more than testing - it is a common language for all team members to use during the development process.

## WHAT IS BEHAVIOR DRIVEN DEVELOPMENT

BDD was introduced by [Dan North](https://dannorth.net/introducing-bdd/). He described it as:

outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

BDD has its own evolution from the days it was born, started by replacing “test” to “should” in unit tests, and moving towards powerful tools like Cucumber and Behat, which made user stories (human readable text) to be executed as an acceptance test.

The idea of story BDD can be narrowed to:

* describe features in a scenario with a formal text
* use examples to make abstract things concrete
* implement each step of a scenario for testing
* write actual code implementing the feature

By writing every feature in User Story format that is automatically executable as a test we ensure that: business, developers, QAs and managers are in the same boat.

BDD encourages exploration and debate in order to formalize the requirements and the features that needs to be implemented by requesting to write the User Stories in a way that everyone can understand.

By making tests to be a part of User Story, BDD allows non-technical personnel to write (or edit) Acceptance tests.

With this procedure we also ensure that everyone in a team knows what has been developed, what has not, what has been tested and what has not.

### UBIQUITOUS LANGUAGE

The ubiquitous language is always referred as common language. That is it’s main benefit. It is not a couple of our business specification’s words, and not a couple of developer’s technical terms. It is a common words and terms that can be understood by people for whom we are building the software and should be understood by developers. Establishing correct communication between this two groups people is vital for building successful project that will fit the domain and fulfill all business needs.

Each feature of a product should be born from a talk between

* business (analysts, product owner)
* developers
* QAs

which are known in BDD as “three amigos”.

Such talks should produce written stories. There should be an actor that doing some things, the feature that should be fulfilled within the story and the result achieved.

We can try to write such simple story:

**As a customer I want to buy several products**

**I put first product with $600 price to my cart**

**And then another one with $1000 price**

**When I go to checkout process**

**I should see that total number of products I want to buy is 2**

**And my order amount is $1600**

As we can see this simple story highlights core concepts that are called contracts. We should fulfill those contracts to model software correctly. But how we can verify that those contracts are being satisfied? [Cucumber](https://cucumber.io/) introduced a special language for such stories called **Gherkin**. Same story transformed to Gherkin will look like this:

**Feature: checkout process**

**In order to buy products**

**As a customer**

**I want to be able to buy several products**

**Scenario:**

**Given I have product with $600 price in my cart**

**And I have product with $1000 price**

**When I go to checkout process**

**Then I should see that total number of products is 2**

**And my order amount is $1600**

Cucumber, Behat, and sure, **Codeception** can execute this scenario step by step as an automated test. Every step in this scenario requires a code which defines it.

## GHERKIN

Let’s learn some more about Gherkin format and then we will see how to execute it with Codeception:

### FEATURES

Whenever you start writing a story you are describing a specific feature of an application, with a set of scenarios and examples describing this feature.

Feature file is written in Gherkin format. Codeception can generate a feature file for you. We will assume that we will use scenarios in feature files for acceptance tests, so feature files to be placed in acceptance suite directory:

**php vendor/bin/codecept g:feature acceptance checkout**

Generated template will look like this:

**Feature: checkout**

**In order to ...**

**As a ...**

**I need to ...**

**Scenario: try checkout**

This template can be fulfilled by setting actor and goals:

**Feature: checkout**

**In order to buy product**

**As a customer**

**I need to be able to checkout the selected products**

Next, we will describe this feature by writing examples for it

#### Scenarios

Scenarios are live examples of feature usage. Inside a feature file it should be written inside a Feature block. Each scenario should contain its title:

**Feature: checkout**

**In order to buy product**

**As a customer**

**I need to be able to checkout the selected products**

**Scenario: order several products**

Scenarios are written in step-by-step manner using Given-When-Then approach. At start, scenario should describe its context with **Given** keyword:

**Given I have product with $600 price in my cart**

**And I have product with $1000 price in my cart**

Here we also use word **And** to extend the Given and not to repeat it in each line.

This is how we described the initial conditions. Next, we perform some action. We use **When** keyword for it:

**When I go to checkout process**

And in the end we are verifying our expectation using **Then** keyword. The action changed the initial given state, and produced some results. Let’s check that those results are what we actually expect.

**Then I should see that total number of products is 2**

**And my order amount is $1600**

We can test this scenario by executing it in dry-run mode. In this mode test won’t be executed (actually, we didn’t define any step for it, so it won’t be executed in any case).

**$ codecept dry-run acceptance checkout.feature**

**checkout: order several products**

**Signature: checkout:order several products**

**Test: tests/acceptance/checkout.feature:order several products**

**Scenario --**

**In order to buy product**

**As a customer**

**I need to be able to checkout the selected products**

**Given i have product with $600 price in my cart**

**And i have product with $1000 price in my cart**

**When i go to checkout process**

**Then i should see that total number of products is 2**

**And my order amount is $1600**

**INCOMPLETE**

**Step definition for `I have product with $600 price in my cart` not found in contexts**

**Step definition for `I have product with $1000 price` not found in contexts**

**Step definition for `I go to checkout process` not found in contexts**

**Step definition for `I should see that total number of products is 2` not found in contexts**

**Step definition for `my order amount is $1600` not found in contexts**

**Run gherkin:snippets to define missing steps**

Besides the scenario steps listed we got the notification that our steps are not defined yet. We can define them easily by executing gherkin:snippets command for the given suite:

**codecept gherkin:snippets acceptance**

This will produce code templates for all undefined steps in all feature files of this suite. Our next step will be to define those steps and transforming feature-file into valid test.

### STEP DEFINITIONS

To match steps from a feature file to PHP code we use annotation which are added to class methods. By default Codeception expects that all methods marked with @Given, @When, @Then annotation. Each annotation should contain a step string.

**/\*\* @Given I am logged as admin \*/**

Steps can also be matched with regex expressions. This way we can make more flexible steps

**/\*\* @Given /I am (logged|authorized) as admin/ \*/**

Please note that regular expressions should start and end with / char. Regex is also used to match parameters and pass them as arguments into methods.

**/\*\***

**\* @Given /I am (?:logged|authorized) as "(\w+)"/**

**\*/**

**function amAuthorized($role)**

**{**

***// logged or authorized does not matter to us***

***// so we added ?: for this capture group***

**}**

Parameters can be also passed in non-regex strings using “:” params placeholder.

**/\*\* @Given I am logged in as :role \*/**

This will match any word (passed in double quotes) or a number passed:

**Given I am logged in as "admin"**

**Given I am logged in as 1**

Steps are defined in Context files. Default context is an actor class, i.e. for acceptance testing suite default context is AcceptanceTester class. However, you can define steps in any classes and include them as contexts. It is useful to define steps in StepObject and PageObject classes.

To list all defined steps run gherkin:steps command:

**codecept gherkin:steps**

## TESTING BEHAVIOR

As it was mentioned, feature files is not just a user story. By writing features in formal language called Gherkin we can execute those scenarios as automated tests. There is no restrictions in the way how those scenarios are supposed to be tested. Tests can be executed at functional, acceptance, or domain level. However, we will concentrate on acceptance or UI tests in current guide.

### ACCEPTANCE TESTING

As we generated snippets for missing steps with gherkin:snippets command, we will define them in AcceptanceTester file.

**class AcceptanceTester extends \Codeception\Actor**

**{**

**use \_generated\AcceptanceTesterActions;**

**/\*\***

**\* @Given I have product with :num1 price in my cart**

**\*/**

**public function iHaveProductWithPriceInMyCart($num1)**

**{**

**throw new \PHPUnit\Framework\IncompleteTestError("Step `I have product with :num1 price in my cart` is not defined");**

**}**

**/\*\***

**\* @When I go to checkout process**

**\*/**

**public function iGoToCheckoutProcess()**

**{**

**throw new \PHPUnit\Framework\IncompleteTestError("Step `I go to checkout process` is not defined");**

**}**

**/\*\***

**\* @Then I should see that total number of products is :num1**

**\*/**

**public function iShouldSeeThatTotalNumberOfProductsIs($num1)**

**{**

**throw new \PHPUnit\Framework\IncompleteTestError("Step `I should see that total number of products is :num1` is not defined");**

**}**

**/\*\***

**\* @Then my order amount is :num1**

**\*/**

**public function myOrderAmountIs($num1)**

**{**

**throw new \PHPUnit\Framework\IncompleteTestError("Step `my order amount is :num1` is not defined");**

**}**

**}**

Please note that :num1 placeholder can be used for strings and numbers (may contain currency sign). In current case :num1 matches $600 and $num1 is assigned to be 600. If you need to receive exact string, wrap the value into quotes: "600$"

By default they throw Incomplete exceptions to ensure test with missing steps won’t be accidentally marked as successful. We will need to implement those steps. As we are in acceptance suite we are probably using [PHPBrowser](https://codeception.com/docs/modules/PhpBrowser) or [WebDriver](https://codeception.com/docs/modules/WebDriver) modules. This means that we can use their methods inside Tester file, as we do with writing tests using $I->. You can use amOnPage, click, see methods inside a step definitions, so each Gherkin scenario step to be extended with basic Codeception steps. Let’s show how it can be implemented in our case:

**class AcceptanceTester extends \Codeception\Actor**

**{**

**use \_generated\AcceptanceTesterActions;**

**/\*\***

**\* @Given I have product with :num1 price in my cart**

**\*/**

**public function iHaveProductWithPriceInMyCart($num1)**

**{**

***// haveRecord method is available in Laravel, Phalcon, Yii modules***

**$productId = $this->haveRecord('Product', ['name' => 'randomProduct'.uniqid(), 'price' => $num1]);**

**$this->amOnPage("/item/$productId");**

**$this->click('Order');**

**}**

**/\*\***

**\* @When I go to checkout process**

**\*/**

**public function iGoToCheckoutProcess()**

**{**

**$this->amOnPage('/checkout');**

**}**

**/\*\***

**\* @Then I should see that total number of products is :num1**

**\*/**

**public function iShouldSeeThatTotalNumberOfProductsIs($num1)**

**{**

**$this->see($num1, '.products-count');**

**}**

**/\*\***

**\* @Then my order amount is :num1**

**\*/**

**public function myOrderAmountIs($num1)**

**{**

**$this->see($num1, '.total');**

**}**

**}**

To make testing more effective we assumed that we are using one of the ActiveRecord frameworks like Laravel, Yii, or Phalcon so we are able to dynamically create records in database with haveRecord method. After that we are opening browser and testing our web pages to see that after selecting those products we really see the price was calculated correctly.

We can dry-run (or run) our feature file to see that Given/When/Then are expanded with substeps:

**Given i have product with $600 price in my cart**

**I have record 'Product',{"name":"randomProduct571fad4f88a04","price":"600"}**

**I am on page "/item/1"**

**I click "Order"**

**And i have product with $1000 price in my cart**

**I have record 'Product',{"name":"randomProduct571fad4f88b14","price":"1000"}**

**I am on page "/item/2"**

**I click "Order"**

**When i go to checkout process**

**I am on page "/checkout"**

**Then i should see that total number of products is 2**

**I see "2",".products-count"**

**And my order amount is $1600**

**I see "1600",".total"**

This way feature file runs just the same as any other Codeception test. Substeps give us detailed information of how the scenario is being executed.

One of the criticism for testing with Gherkin was that only technical team were aware of how the test scenario is executed. This could have lead to false-positive tests. Developers could have used empty steps for scenarios (or irrelevant ones) and produced invalid tests for valid scenarios. Codeception brings communication to a next level, everyone in a team can understand what happens on a lower (technical) level. Scenario expanding to substeps shows the actual test execution process. Anyone in a team can read the output, and invest their efforts into improving the test suite.

## ADVANCED GHERKIN

Let’s improve our BDD suite by using the advanced features of Gherkin language.

### BACKGROUND

If a group of scenarios have the same initial steps, let’s that for dashboard we always need to be logged in as administrator. We can use Background section to do the required preparations and not to repeat same steps across scenarios.

**Feature: Dashboard**

**In order to view current state of business**

**As an owner**

**I need to be able to see reports on dashboard**

**Background:**

**Given I am logged in as administrator**

**And I open dashboard page**

Steps in background are defined the same way as in scenarios.

### TABLES

Scenarios can become more descriptive when you represent repeating data as tables. Instead of writing several steps “I have product with :num1 $ price in my cart” we can have one step with multiple values in it.

**Given i have products in my cart**

**| name | category | price |**

**| Harry Potter | Books | 5 |**

**| iPhone 5 | Smartphones | 1200 |**

**| Nuclear Bomb | Weapons | 100000 |**

Tables is a recommended ways to pass arrays into test scenarios. Inside a step definition data is stored in argument passed as \Behat\Gherkin\Node\TableNode instance.

**/\*\***

**\* @Given i have products in my cart**

**\*/**

**public function iHaveProductsInCart(\Behat\Gherkin\Node\TableNode $products)**

**{**

***// iterate over all rows***

**foreach ($node->getRows() as $index => $row) {**

**if ($index === 0) { *// first row to define fields***

**$keys = $row;**

**continue;**

**}**

**$this->haveRecord('Product', array\_combine($keys, $row));**

**}**

**}**

### EXAMPLES

In case scenarios represent the same logic but differ on data, we can use Scenario Outline to provide different examples for the same behavior. Scenario outline is just like a basic scenario with some values replaced with placeholders, which are filled from a table. Each set of values is executed as a different test.

**Scenario Outline: order discount**

**Given I have product with price <price>$ in my cart**

**And discount for orders greater than $20 is 10 %**

**When I go to checkout**

**Then I should see overall price is "<total>" $**

**Examples:**

**| price | total |**

**| 10 | 10 |**

**| 20 | 20 |**

**| 21 | 18.9 |**

**| 30 | 27 |**

**| 50 | 45 |**

### LONG STRINGS

Text values inside a scenarios can be set inside a """ block:

**Then i see in file "codeception.yml"**

**"""**

**paths:**

**tests: tests**

**log: tests/\_output**

**data: tests/\_data**

**helpers: tests/\_support**

**envs: tests/\_envs**

**"""**

This string is passed as a standard PHP string parameter

**/\*\***

**\* @Then i see in file :filename**

**\*/**

**public function seeInFile($fileName, $fileContents)**

**{**

***// note: module "Asserts" is enabled in this suite***

**if (!file\_exists($fileName)) {**

**$this->fail("File $fileName not found");**

**}**

**$this->assertEquals(file\_get\_contents($fileName), $fileContents);**

**}**

### TAGS

Gherkin scenarios and features can contain tags marked with @. Tags are equal to groups in Codeception. This way if you define a feature with @important tag, you can execute it inside important group by running:

**codecept run -g important**

Tag should be placed before Scenario: or before Feature: keyword. In the last case all scenarios of that feature will be added to corresponding group.

## CONFIGURATION

As we mentioned earlier, steps should be defined inside context classes. By default all the steps are defined inside an Actor class, for instance, AcceptanceTester. However, you can include more contexts. This can be configured inside global codeception.yml or suite configuration file:

**gherkin:**

**contexts:**

**default:**

**- AcceptanceTester**

**- AdditionalSteps**

AdditionalSteps file should be accessible by autoloader and can be created by Codeception\Lib\Di. This means that practically any class can be a context. If a class receives an actor class in constructor or in \_inject method, DI can inject it into it.

**class AdditionalSteps**

**{**

**protected $I;**

**function \_\_construct(AcceptanceTester $I)**

**{**

**$this->I = $I;**

**}**

**/\*\***

**\* @When I do something**

**\*/**

**function additionalActions()**

**{**

**}**

**}**

This way PageObjects, Helpers and StepObjects can become contexts as well. But more preferable to include context classes by their tags or roles.

If you have Step\Admin class which defines only admin steps, it is a good idea to use it as context for all features containing with “As an admin”. In this case “admin” is a role and we can configure it to use additional context.

**gherkin:**

**contexts:**

**role:**

**admin:**

**- "Step\Admin"**

Contexts can be attached to tags as well. This may be useful if you want to redefine steps for some scenarios. Let’s say we want to bypass login steps for some scenarios loading already defined session. In this case we can create Step\FastLogin class with redefined step “I am logged in as”.

**gherkin:**

**contexts:**

**tag:**

**fastlogin:**

**- "Step\FastLogin"**

Contexts can be autoloaded as well:

**gherkin:**

**contexts:**

**path: tests/\_support/Steps**

**namespace\_prefix: Steps**

**default:**

**- AcceptanceTester**

This will load all context from the given path and prefix it with the given namespace.

## MIGRATING FROM BEHAT

While Behat is a great tool for Behavior Driven Development, you still may prefer to use Codeception as your primary testing framework. In case you want to unify all your tests (unit/functional/acceptance), and make them be executed with one runner, Codeception is a good choice. Also Codeception provides rich set of well-maintained modules for various testing backends like Selenium Webdriver, Symfony, Laravel, etc.

If you decided to run your features with Codeception, we recommend to start with symlinking your features directory into one of the test suites:

**ln -s $PWD/features tests/acceptance**

Then you will need to implement all step definitions. Run gherkin:snippets to generate stubs for them. By default it is recommended to place step definitions into actor class (Tester) and use its methods for steps implementation.

## TESTS VS FEATURES

It is common to think that BDD scenario is equal to test. But it’s actually not. Not every test should be described as a feature. Not every test is written to test real business value. For instance, regression tests or negative scenario tests are not bringing any value to business. Business analysts don’t care about scenario reproducing bug #13, or what error message is displayed when user tries to enter wrong password on login screen. Writing all the tests inside a feature files creates informational overflow.

In Codeception you can combine tests written in Gherkin format with tests written in Cept/Cest/Test formats. This way you can keep your feature files compact with minimal set of scenarios, and write regular tests to cover all cases.

Corresponding features and tests can be attached to the same group. And what is more interesting, you can make tests to depend on feature scenarios. Let’s say we have login.feature file with “Log regular user” scenario in it. In this case you can specify that every test which requires login to pass to depend on “Log regular user” scenario:

**@depends login:Log regular user**

Inside @depends block you should use test signature. Execute your feature with dry-run to see signatures for all scenarios in it. By marking tests with @depends you ensure that this test won’t be executed before the test it depends on.

## CONCLUSIONS

If you like the concept of Behavior Driven Development or prefer to keep test scenarios in human readable format, Codeception allows you to write and execute scenarios in Gherkin. Feature files is just another test format inside Codeception, so it can be combined with Cept and Cest files inside the same suite. Steps definitions of your scenarios can use all the power of Codeception modules, PageObjects, and StepObjects.

* **Next Chapter:**[**Customization >**](https://codeception.com/docs/08-Customization)
* **Previous Chapter:**[**< AdvancedUsage**](https://codeception.com/docs/07-AdvancedUsage)

# CUSTOMIZATION

In this chapter we will explain how you can extend and customize the file structure and test execution routines.

### NAMESPACES

To avoid naming conflicts between Actor classes and Helper classes, they should be separated into namespaces. To create test suites with namespaces you can add --namespace option to the bootstrap command:

**php vendor/bin/codecept bootstrap --namespace frontend**

This will bootstrap a new project with the namespace: frontend parameter in the codeception.yml file. Helpers will be in the frontend\Codeception\Module namespace and Actor classes will be in the frontend namespace.

Once each of your applications (bundles) has its own namespace and different Helper or Actor classes, you can execute all the tests in a single runner. Run the Codeception tests as usual, using the meta-config we created earlier:

**php vendor/bin/codecept run**

This will launch the test suites for all three applications and merge the reports from all of them. This is very useful when you run your tests on a Continuous Integration server and you want to get a single report in JUnit and HTML format. The code coverage report will be merged too.

If you want to run a specific suite from the application you can execute:

**vendor/bin/codecept run unit -c frontend**

Where unit is the name of suite and the -c option specifies the path to the codeception.yml configuration file to use. In this example we will assume that there is frontend/codeception.yml configuration file and that we will execute the unit tests for only that app.

## BOOTSTRAP

To prepare environment for testing you can execute custom PHP script before all tests or just before a specific suite. This way you can initialize autoloader, check availability of a website, etc.

### GLOBAL BOOTSTRAP

To run bootstrap script before all suites place it in tests directory (absolute paths supported as well). Then set a bootstrap config key in codeception.yml:

**yml**

***# file will be loaded from tests/bootstrap.php***

**bootstrap: bootstrap.php**

### SUITE BOOTSTRAP

To run a script for a specific suite, place it into the suite directory and add to suite config:

**yml**

***# inside <suitename>.suite.yml***

***# file will be loaded from tests/<suitename>/bootstrap.php***

**bootstrap: bootstrap.php**

### ON FLY BOOTSTRAP

Bootstrap script can be executed with --bootstrap option for codecept run command:

**vendor/bin/codecept run --bootstrap bootstrap.php**

In this case, bootstrap script will be executed before the Codeception is initialized. Bootstrap script should be located in current working directory or by an absolute path.

Bootstrap is a classical way to run custom PHP code before your tests. However, we recommend you to use Extensions instead of bootstrap scripts for better flexibility. If you need configuration, conditional enabling or disabling bootstrap script, extensions should work for you better.

## EXTENSION

Codeception has limited capabilities to extend its core features. Extensions are not supposed to override current functionality, but can be useful if you are an experienced developer and you want to hook into the testing flow.

By default, one RunFailed Extension is already enabled in your global codeception.yml. It allows you to rerun failed tests by using the -g failed option:

**vendor/bin/codecept run -g failed**

Codeception comes with bundled extensions located in ext directory. For instance, you can enable the Logger extension to log the test execution with Monolog:

**extensions:**

**enabled:**

**- Codeception\Extension\RunFailed *# default extension***

**- Codeception\Extension\Logger: *# enabled extension***

**max\_files: 5 *# logger configuration***

But what are extensions, anyway? Basically speaking, extensions are nothing more than event listeners based on the [Symfony Event Dispatcher](https://symfony.com/doc/current/components/event_dispatcher/introduction.html) component.

### EVENTS

Here are the events and event classes. The events are listed in the order in which they happen during execution. All listed events are available as constants in Codeception\Events class.

| **Event** | **When?** | **Triggered by** |
| --- | --- | --- |
| suite.before | Before suite is executed | [Suite, Settings](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/SuiteEvent.php) |
| test.start | Before test is executed | [Test](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/TestEvent.php) |
| test.before | At the very beginning of test execution | [Codeception Test](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/TestEvent.php) |
| step.before | Before step | [Step](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/StepEvent.php) |
| step.after | After step | [Step](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/StepEvent.php) |
| step.fail | After failed step | [Step](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/StepEvent.php) |
| test.fail | After failed test | [Test, Fail](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/FailEvent.php) |
| test.error | After test ended with error | [Test, Fail](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/FailEvent.php) |
| test.incomplete | After executing incomplete test | [Test, Fail](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/FailEvent.php) |
| test.skipped | After executing skipped test | [Test, Fail](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/FailEvent.php) |
| test.success | After executing successful test | [Test](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/TestEvent.php) |
| test.after | At the end of test execution | [Codeception Test](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/TestEvent.php) |
| test.end | After test execution | [Test](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/TestEvent.php) |
| suite.after | After suite was executed | [Suite, Result, Settings](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/SuiteEvent.php) |
| test.fail.print | When test fails are printed | [Test, Fail](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/FailEvent.php) |
| result.print.after | After result was printed | [Result, Printer](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Event/PrintResultEvent.php) |

There may be some confusion between test.start/test.before and test.after/test.end. The start and end events are triggered by PHPUnit, but the before and after events are triggered by Codeception. Thus, when you are using classical PHPUnit tests (extended from PHPUnit\Framework\TestCase), the before/after events won’t be triggered for them. During the test.before event you can mark a test as skipped or incomplete, which is not possible in test.start. You can learn more from [Codeception internal event listeners](https://github.com/Codeception/Codeception/tree/4.0/src/Codeception/Subscriber).

The extension class itself is inherited from Codeception\Extension:

**use \Codeception\Events;**

**class MyCustomExtension extends \Codeception\Extension**

**{**

***// list events to listen to***

***// Codeception\Events constants used to set the event***

**public static $events = array(**

**Events::SUITE\_AFTER => 'afterSuite',**

**Events::TEST\_BEFORE => 'beforeTest',**

**Events::STEP\_BEFORE => 'beforeStep',**

**Events::TEST\_FAIL => 'testFailed',**

**Events::RESULT\_PRINT\_AFTER => 'print',**

**);**

***// methods that handle events***

**public function afterSuite(\Codeception\Event\SuiteEvent $e) {}**

**public function beforeTest(\Codeception\Event\TestEvent $e) {}**

**public function beforeStep(\Codeception\Event\StepEvent $e) {}**

**public function testFailed(\Codeception\Event\FailEvent $e) {}**

**public function print(\Codeception\Event\PrintResultEvent $e) {}**

**}**

By implementing event handling methods you can listen for events and even update passed objects. Extensions have some basic methods you can use:

* write - prints to the screen
* writeln - prints to the screen with a new-line character at the end
* getModule - allows you to access a module
* hasModule - checks if a module is enabled
* getModuleNames - list all enabled modules
* \_reconfigure - can be implemented instead of overriding the constructor

### ENABLING EXTENSION

Once you’ve implemented a simple extension class, you can require it in tests/\_bootstrap.php, load it with Composer’s autoloader defined in composer.json, or store the class inside tests/\_supportdir.

You can then enable it in codeception.yml

**extensions:**

**enabled: [MyCustomExtension]**

Extensions can also be enabled per suite inside suite configs (like acceptance.suite.yml) and for a specific environment.

To enable extension dynamically, execute the run command with --ext option. Provide a class name as a parameter:

**php vendor/bin/codecept run --ext MyCustomExtension**

**php vendor/bin/codecept run --ext "\My\Extension"**

If a class is in a Codeception\Extension namespace you can skip it and provide only a shortname. So Recorder extension can be started like this:

**php vendor/bin/codecept run --ext Recorder**

### CONFIGURING EXTENSION

In the extension, you can access the currently passed options via the options property. You also can access the global configuration via the \Codeception\Configuration::config() method. If you want to have custom options for your extension, you can pass them in the codeception.yml file:

**extensions:**

**enabled: [MyCustomExtension]**

**config:**

**MyCustomExtension:**

**param: value**

The passed in configuration is accessible via the config property: $this->config['param'].

Check out a very basic extension [Notifier](https://github.com/Codeception/Notifier).

### CUSTOM COMMANDS

You can add your own commands to Codeception.

Your custom commands have to implement the interface Codeception\CustomCommandInterface, because there has to be a function to get the name of the command.

You have to register your command in the file codeception.yml:

**extensions:**

**commands: [Project\Command\MyCustomCommand]**

If you want to activate the Command globally, because you are using more then one codeception.yml file, you have to register your command in the codeception.dist.yml in the root folder of your project.

Please see the [complete example](https://github.com/Codeception/Codeception/blob/4.0/tests/data/register_command/examples/MyCustomCommand.php)

## GROUP OBJECTS

Group Objects are extensions listening for the events of tests belonging to a specific group. When a test is added to a group:

**/\*\***

**\* @group admin**

**\*/**

**public function testAdminCreatingNewBlogPost(\AcceptanceTester $I)**

**{**

**}**

This test will trigger the following events:

* test.before.admin
* step.before.admin
* step.after.admin
* test.success.admin
* test.fail.admin
* test.after.admin

A group object is built to listen for these events. It is useful when an additional setup is required for some of your tests. Let’s say you want to load fixtures for tests that belong to the admin group:

**namespace Group;**

**class Admin extends \Codeception\GroupObject**

**{**

**public static $group = 'admin';**

**public function \_before(\Codeception\Event\TestEvent $e)**

**{**

**$this->writeln('inserting additional admin users...');**

**$db = $this->getModule('Db');**

**$db->haveInDatabase('users', ['name' => 'bill', 'role' => 'admin']);**

**$db->haveInDatabase('users', ['name' => 'john', 'role' => 'admin']);**

**$db->haveInDatabase('users', ['name' => 'mark', 'role' => 'banned']);**

**}**

**public function \_after(\Codeception\Event\TestEvent $e)**

**{**

**$this->writeln('cleaning up admin users...');**

***// ...***

**}**

**}**

GroupObjects can also be used to update the module configuration before running a test. For instance, for nocleanup group we prevent Doctrine2 module from wrapping test into transaction:

**public static $group = 'nocleanup';**

**public function \_before(\Codeception\Event\TestEvent $e)**

**{**

**$this->getModule('Doctrine2')->\_reconfigure(['cleanup' => false]);**

**}**

A group class can be created with php vendor/bin/codecept generate:group groupname command. Group classes will be stored in the tests/\_support/Group directory.

A group class can be enabled just like you enable an extension class. In the file codeception.yml:

**extensions:**

**enabled: [Group\Admin]**

Now the Admin group class will listen for all events of tests that belong to the admin group.

## STEP DECORATORS

Actor classes include generated steps taken from corresponding modules and helpers. You can introduce wrappers for those steps by using step decorators.

Step decorators are used to implement conditional assertions. When enabled, conditional assertions take all method prefixed by see or dontSee and introduce new steps prefixed with canSee and cantSee. Contrary to standard assertions those assertions won’t stop test on failure. This is done by wrapping action into try/catch blocks.

List of available step decorators:

* [ConditionalAssertion](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Step/ConditionalAssertion.php) - failed assertion will be logged, but test will continue.
* [TryTo](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Step/TryTo.php) - failed action will be ignored.
* [Retry](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Step/Retry.php) - failed action will be retried automatically.

Step decorators can be added to suite config inside steps block:

**yml**

**step\_decorators:**

**- Codeception/Step/TryTo**

**- Codeception/Step/Retry**

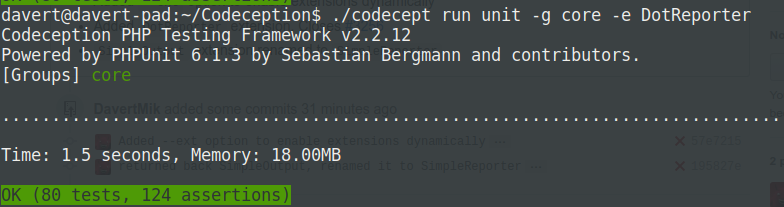
**- Codeception/Step/ConditionalAssertion**

You can introduce your own step decorators. Take a look into sample decorator classes and create your own class which implements Codeception\Step\GeneratedStep interface. A class should provide getTemplate method which returns a code block and variables passed into a template. Make your class accessible by autoloader and you can have your own step decorators working.

## CUSTOM REPORTERS

Alternative reporters can be implemented as extension. There are [DotReporter](https://codeception.com/extensions#DotReporter) and [SimpleReporter](https://codeception.com/extensions#SimpleReporter) extensions included. Use them to change output or use them as an example to build your own reporter. They can be easily enabled with --ext option

**php vendor/bin/codecept run --ext DotReporter**



If you want to use it as default reporter enable it in codeception.yml.

But what if you need to change the output format of the XML or JSON results triggered with the --xml or --json options? Codeception uses PHPUnit printers and overrides them. If you need to customize one of the standard reporters you can override them too. If you are thinking on implementing your own reporter you should add a reporters section to codeception.yml and override one of the standard printer classes with one of your own:

**reporters:**

**xml: Codeception\PHPUnit\Log\JUnit**

**html: Codeception\PHPUnit\ResultPrinter\HTML**

**report: Codeception\PHPUnit\ResultPrinter\Report**

All PHPUnit printers implement the [PHPUnit\_Framework\_TestListener](https://phpunit.de/manual/current/en/extending-phpunit.html#extending-phpunit.PHPUnit_Framework_TestListener) interface. It is recommended to read the code of the original reporter before overriding it.

## INSTALLATION TEMPLATES

Codeception setup can be customized for the needs of your application. If you build a distributable application and you have a personalized configuration you can build an Installation template which will help your users to start testing on their projects.

Codeception has built-in installation templates for

* [Acceptance tests](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Template/Acceptance.php)
* [Unit tests](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Template/Unit.php)
* [REST API tests](https://github.com/Codeception/Codeception/blob/4.0/src/Codeception/Template/Api.php)

They can be executed with init command:

**php vendor/bin/codecept init Acceptance**

To init tests in specific folder use --path option:

**php vendor/bin/codecept init Acceptance --path acceptance\_tests**

You will be asked several questions and then config files will be generated and all necessary directories will be created. Learn from the examples above to build a custom Installation Template. Here are the basic rules you should follow:

* Templates should be inherited from [Codeception\InitTemplate](https://codeception.com/docs/reference/InitTemplate) class and implement setup method.
* Template class should be placed in Codeception\Template namespace so Codeception could locate them by class name
* Use methods like say, saySuccess, sayWarning, sayError, ask, to interact with a user.
* Use createDirectoryFor, createEmptyDirectory methods to create directories
* Use createHelper, createActor methods to create helpers and actors.
* Use [Codeception generators](https://github.com/Codeception/Codeception/tree/4.0/src/Codeception/Lib/Generator) to create other support classes.

## ONE RUNNER FOR MULTIPLE APPLICATIONS

If your project consists of several applications (frontend, admin, api) or you are using the Symfony framework with its bundles, you may be interested in having all tests for all applications (bundles) executed in one runner. In this case you will get one report that covers the whole project.

Place the codeception.yml file into the root folder of your project and specify the paths to the other codeception.yml configurations that you want to include:

**include:**

**- frontend/src/\*Bundle**

**- admin**

**- api/rest**

**paths:**

**output: \_output**

**settings:**

**colors: false**

You should also specify the path to the log directory, where the reports and logs will be saved.

Wildcards (\*) can be used to specify multiple directories at once.

## CONCLUSION

Each feature mentioned above may help dramatically when using Codeception to automate the testing of large projects, although some features may require advanced knowledge of PHP. There is no “best practice” or “use cases” when we talk about groups, extensions, or other powerful features of Codeception. If you see you have a problem that can be solved using these extensions, then give them a try.

* **Next Chapter:**[**Data >**](https://codeception.com/docs/09-Data)
* **Previous Chapter:**[**< BDD**](https://codeception.com/docs/07-BDD)

# WORKING WITH DATA

Tests should not affect each other. That’s a rule of thumb. When tests interact with a database, they may change the data inside it, which would eventually lead to data inconsistency. A test may try to insert a record that has already been inserted, or retrieve a deleted record. To avoid test failures, the database should be brought back to its initial state before each test. Codeception has different methods and approaches to get your data cleaned.

This chapter summarizes all of the notices on clean-ups from the previous chapters and suggests the best strategies of how to choose data storage backends.

When we decide to clean up a database, we should make this cleaning as fast as possible. Tests should always run fast. Rebuilding the database from scratch is not the best way, but might be the only one. In any case, you should use a special test database for testing. **Do not ever run tests on development or production databases!**

## DB

Codeception has a Db module, which takes on most of the tasks of database interaction.

**modules:**

**config:**

**Db:**

**dsn: 'PDO DSN HERE'**

**user: 'root'**

**password:**

Use [module parameters](https://codeception.com/docs/06-ModulesAndHelpers#Dynamic-Configuration-With-Params) to set the database credentials from environment variables or from application configuration files.

Db module can cleanup database between tests by loading a database dump. This can be done by parsing SQL file and executing its commands using current connection

**modules:**

**config:**

**Db:**

**dsn: 'PDO DSN HERE'**

**user: 'root'**

**password:**

**dump: tests/\_data/your-dump-name.sql**

**cleanup: true *# reload dump between tests***

**populate: true *# load dump before all tests***

Alternatively an external tool (like mysql client, or pg\_restore) can be used. This approach is faster and won’t produce parsing errors while loading a dump. Use populator config option to specify the command. For MySQL it can look like this:

**modules:**

**enabled:**

**- Db:**

**dsn: 'mysql:host=localhost;dbname=testdb'**

**user: 'root'**

**password: ''**

**cleanup: true *# run populator before each test***

**populate: true *# run populator before all test***

**populator: 'mysql -u $user $dbname < tests/\_data/dump.sql'**

See the [Db module reference](https://codeception.com/docs/modules/Db#SQL-data-dump) for more examples.

To ensure database dump is loaded before all tests add populate: true. To clean current database and reload dump between tests use cleanup: true.

A full database clean-up can be painfully slow if you use large database dumps. It is recommended to do more data testing on the functional and integration levels, this way you can get performance bonuses from using ORM.

In acceptance tests, your tests are interacting with the application through a web server. This means that the test and the application work with the same database. You should provide the same credentials in the Db module that your application uses, then you can access the database for assertions (seeInDatabase actions) and to perform automatic clean-ups.

The Db module provides actions to create and verify data inside a database.

If you want to create a special database record for one test, you can use [haveInDatabase](https://codeception.com/docs/modules/Db#haveInDatabase) method of Db module:

**$I->haveInDatabase('posts', [**

**'title' => 'Top 10 Testing Frameworks',**

**'body' => '1. Codeception'**

**]);**

**$I->amOnPage('/posts');**

**$I->see('Top 10 Testing Frameworks');**

haveInDatabase inserts a row with the provided values into the database. All added records will be deleted at the end of the test.

If you want to check that a table record was created use [seeInDatabase](https://codeception.com/docs/modules/Db#haveInDatabase) method:

**$I->amOnPage('/posts/1');**

**$I->fillField('comment', 'This is nice!');**

**$I->click('Submit');**

**$I->seeInDatabase('comments', ['body' => 'This is nice!']);**

See the module [reference](https://codeception.com/docs/modules/Db) for other methods you can use for database testing.

There are also modules for [MongoDb](https://codeception.com/docs/modules/MongoDb), [Redis](https://codeception.com/docs/modules/Redis), and [Memcache](https://codeception.com/docs/modules/Memcache) which behave in a similar manner.

### SEQUENCE

If the database clean-up takes too long, you can follow a different strategy: create new data for each test. This way, the only problem you might face is duplication of data records. [Sequence](https://codeception.com/docs/modules/Sequence) was created to solve this. It provides the sq() function which generates unique suffixes for creating data in tests.

## ORM MODULES

Your application is most likely using object-relational mapping (ORM) to work with the database. In this case, Codeception allows you to use the ORM methods to work with the database, instead of accessing the database directly. This way you can work with models and entities of a domain, and not on tables and rows.

By using ORM in functional and integration tests, you can also improve performance of your tests. Instead of cleaning up the database after each test, the ORM module will wrap all the database actions into transactions and roll it back at the end. This way, no actual data will be written to the database. This clean-up strategy is enabled by default, you can disable it by setting cleanup: false in the configuration of any ORM module.

### ACTIVERECORD

Popular frameworks like Laravel, Yii, and Phalcon include an ActiveRecord data layer by default. Because of this tight integration, you just need to enable the framework module, and use its configuration for database access.

Corresponding framework modules provide similar methods for ORM access:

* haveRecord
* seeRecord
* dontSeeRecord
* grabRecord

They allow you to create and check data by model name and field names in the model. Here is the example in Laravel:

***// create record and get its id***

**$id = $I->haveRecord('posts', ['body' => 'My first blogpost', 'user\_id' => 1]);**

**$I->amOnPage('/posts/'.$id);**

**$I->see('My first blogpost', 'article');**

***// check record exists***

**$I->seeRecord('posts', ['id' => $id]);**

**$I->click('Delete');**

***// record was deleted***

**$I->dontSeeRecord('posts', ['id' => $id]);**

Laravel5 module provides the method have which uses the [factory](https://laravel.com/docs/5.8/database-testing#generating-factories) method to generate models with fake data.

If you want to use ORM for integration testing only, you should enable the framework module with only the ORM part enabled:

**modules:**

**enabled:**

**- Laravel5:**

**- part: ORM**

**modules:**

**enabled:**

**- Yii2:**

**- part: ORM**

This way no web actions will be added to $I object.

If you want to use ORM to work with data inside acceptance tests, you should also include only the ORM part of a module. Please note that inside acceptance tests, web applications work inside a webserver, so any test data can’t be cleaned up by rolling back transactions. You will need to disable cleaning up, and use the Db module to clean the database up between tests. Here is a sample config:

**modules:**

**enabled:**

**- WebDriver:**

**url: http://localhost**

**browser: firefox**

**- Laravel5:**

**cleanup: false**

**- Db**

### DOCTRINE

Doctrine is also a popular ORM, unlike some others it implements the DataMapper pattern and is not bound to any framework. The [Doctrine2](https://codeception.com/docs/modules/Doctrine2) module requires an EntityManager instance to work with. It can be obtained from a Symfony framework or Zend Framework (configured with Doctrine):

**modules:**

**enabled:**

**- Symfony**

**- Doctrine2:**

**depends: Symfony**

**modules:**

**enabled:**

**- ZF2**

**- Doctrine2:**

**depends: ZF2**

If no framework is used with Doctrine you should provide the connection\_callback option with a valid callback to a function which returns an EntityManager instance.

Doctrine2 also provides methods to create and check data:

* haveInRepository
* grabFromRepository
* grabEntitiesFromRepository
* seeInRepository
* dontSeeInRepository

### DATAFACTORY

Preparing data for testing is a very creative, although boring, task. If you create a record, you need to fill in all the fields of the model. It is much easier to use [Faker](https://github.com/fzaninotto/Faker) for this task, which is more effective to set up data generation rules for models. Such a set of rules is called factories and are provided by the [DataFactory](https://codeception.com/docs/modules/DataFactory) module.

Once configured, it can create records with ease:

***// creates a new user***

**$user\_id = $I->have('App\Model\User');**

***// creates 3 posts***

**$I->haveMultiple('App\Model\Post', 3);**

Created records will be deleted at the end of a test. The DataFactory module only works with ORM, so it requires one of the ORM modules to be enabled:

**modules:**

**enabled:**

**- Yii2:**

**configFile: path/to/config.php**

**- DataFactory:**

**depends: Yii2**

**modules:**

**enabled:**

**- Symfony**

**- Doctrine2:**

**depends: Symfony**

**- DataFactory:**

**depends: Doctrine2**

DataFactory provides a powerful solution for managing data in integration/functional/acceptance tests. Read the [full reference](https://codeception.com/docs/modules/DataFactory) to learn how to set this module up.

## TESTING DYNAMIC DATA WITH SNAPSHOTS

What if you deal with data which you don’t own? For instance, the page look depends on number of categories in database, and categories are set by admin user. How would you test that the page is still valid?

There is a way to get it tested as well. Codeception allows you take a snapshot of a data on first run and compare with on next executions. This principle is so general that it can work for testing APIs, items on a web page, etc.

Let’s check that list of categories on a page is the same it was before.  
Create a snapshot class:

**vendor/bin/codecept g:snapshot Categories**

Inject an actor class via constructor and implement fetchData method which should return a data set from a test.

**namespace Snapshot;**

**class Categories extends \Codeception\Snapshot**

**{**

**/\*\* @var \AcceptanceTester \*/**

**protected $i;**

**public function \_\_construct(\AcceptanceTester $I)**

**{**

**$this->i = $I;**

**}**

**protected function fetchData()**

**{**

***// fetch texts from all 'a.category' elements on a page***

**return $this->i->grabMultiple('a.category');**

**}**

**}**

Inside a test you can inject the snapshot class and call assert method on it:

**public function testCategoriesAreTheSame(\AcceptanceTester $I, \Snapshot\Categories $snapshot)**

**{**

**$I->amOnPage('/categories');**

***// if previously saved array of users does not match current set, test will fail***

***// to update data in snapshot run test with --debug flag***

**$snapshot->assert();**

**}**

On the first run, data will be obtained via fetchData method and saved to tests/\_data directory in json format. On next execution the obtained data will be compared with previously saved snapshot.

To update a snapshot with a new data run tests in --debug mode.

By default Snapshot uses assertEquals assertion, however this can be customized by overriding assertData method.

### FAILED ASSERTION OUTPUT

The assertion performed by assertData will not display the typical diff output from assertEquals or any customized failed assertion. To have the diff displayed when running tests, you can call the snapshot method shouldShowDiffOnFail:

**public function testCategoriesAreTheSame(\AcceptanceTester $I, \Snapshot\Categories $snapshot)**

**{**

**$I->amOnPage('/categories');**

***// I want to see the diff in case the snapshot data changes***

**$snapshot->shouldShowDiffOnFail();**

**$snapshot->assert();**

**}**

If ever needed, the diff output can also be omitted by calling shouldShowDiffOnFail(false).

### WORKING WITH DIFFERENT DATA FORMATS

By default, all snapshot files are stored in json format, so if you have to work with different formats, neither the diff output or the snapshot file data will be helpful. To fix this, you can call the snapshot method shouldSaveAsJson(false) and set the file extension by calling setSnapshotFileExtension():

**public function testCategoriesAreTheSame(\AcceptanceTester $I, \Snapshot\Categories $snapshot)**

**{**

***// I fetch an HTML page***

**$I->amOnPage('/categories.html');**

***// I want to see the diff in case the snapshot data changes***

**$snapshot->shouldSaveAsJson(false);**

**$snapshot->setSnapshotFileExtension('html');**

**$snapshot->assert();**

**}**

The snapshot file will be stored without encoding it to json format, and with the .html extension.

Beware that this option will not perform any changes in the data returned by fetchData, and store it as it is.

## CONCLUSION

Codeception also assists the developer when dealing with data. Tools for database population and cleaning up are bundled within the Db module. If you use ORM, you can use one of the provided framework modules to operate with database through a data abstraction layer, and use the DataFactory module to generate new records with ease.

* **Next Chapter:**[**APITesting >**](https://codeception.com/docs/10-APITesting)
* **Previous Chapter:**[**< Customization**](https://codeception.com/docs/08-Customization)

# API TESTING

The same way we tested a web site, Codeception allows you to test web services. They are very hard to test manually, so it’s a really good idea to automate web service testing. We have SOAP and REST as standards, which are represented in corresponding modules, which we will cover in this chapter.

You should start by creating a new test suite, (which was not provided by the bootstrap command). We recommend calling it **api** and using the ApiTester class for it.

**php vendor/bin/codecept generate:suite api**

We will put all the api tests there.

## REST API

The REST web service is accessed via HTTP with standard methods: GET, POST, PUT, DELETE. They allow users to receive and manipulate entities from the service. Accessing a WebService requires an HTTP client, so for using it you need the module PhpBrowser or one of framework modules set up. For example, we can use the Symfony module for Symfony2 applications in order to ignore web server and test web service internally.

Configure modules in api.suite.yml:

**actor: ApiTester**

**modules:**

**enabled:**

**- REST:**

**url: http://serviceapp/api/v1/**

**depends: PhpBrowser**

The REST module will connect to PhpBrowser according to this configuration. Depending on the web service we may deal with XML or JSON responses. Codeception handles both data formats well, however If you don’t need one of them, you can explicitly specify that the JSON or XML parts of the module will be used:

**actor: ApiTester**

**modules:**

**enabled:**

**- REST:**

**url: http://serviceapp/api/v1/**

**depends: PhpBrowser**

**part: Json**

API tests can be functional and be executed using Symfony, Laravel5, Zend, or any other framework module. You will need slightly update configuration for it:

**actor: ApiTester**

**modules:**

**enabled:**

**- REST:**

**url: /api/v1/**

**depends: Laravel5**

Once we have configured our new testing suite, we can create the first sample test:

**php vendor/bin/codecept generate:cest api CreateUser**

It will be called CreateUserCest.php. We need to implement a public method for each test. Let’s make createUserViaAPI to test creation of a user via the REST API.

**class CreateUserCest**

**{**

***// tests***

**public function createUserViaAPI(\ApiTester $I)**

**{**

**$I->amHttpAuthenticated('service\_user', '123456');**

**$I->haveHttpHeader('Content-Type', 'application/x-www-form-urlencoded');**

**$I->sendPost('/users', [**

**'name' => 'davert',**

**'email' => 'davert@codeception.com'**

**]);**

**$I->seeResponseCodeIs(\Codeception\Util\HttpCode::OK); *// 200***

**$I->seeResponseIsJson();**

**$I->seeResponseContains('{"result":"ok"}');**

**}**

**}**

We can use HTTP code constants from Codeception\Util\HttpCode instead of numeric values to check response code in seeResponseCodeIs and dontSeeResponseCodeIs methods.

Let’s see what the test consist of.

### AUTHORIZATION

To authorize requests to external resources, usually provider requires you to authorize using headers. Additional headers can be set before request using haveHttpHeader command:

**$I->haveHttpHeader('api\_key', 'special-key');**

For common authorization patterns use one of the following methods:

* amAWSAuthenticated
* amBearerAuthenticated
* amDigestAuthenticated
* amHttpAuthenticated
* amNTLMAuthenticated

### SENDING REQUESTS

The real action in a test happens only when a request is sent. Before a request you may provide additional http headers which will be used in a next request to set authorization or expected content format.

**$I->haveHttpHeader('accept', 'application/json');**

**$I->haveHttpHeader('content-type', 'application/json');**

When headers are set, you can send a request. To obtain data use sendGet:

***// pass in query params in second argument***

**$I->sendGet('/posts', [ 'status' => 'pending' ]);**

**$I->seeResponseCodeIs(200);**

**$I->seeResponseIsJson();**

sendGet won’t return any value. However, you can access data from a response and perform assertions using other available methods of REST module.

To create or update data you can use other common methods:

* sendPost
* sendPut
* sendDelete
* sendPatch

### JSON STRUCTURE VALIDATION

If we expect a JSON response to be received we can check its structure with [JSONPath](http://goessner.net/articles/JsonPath/). It looks and sounds like XPath but is designed to work with JSON data, however we can convert JSON into XML and use XPath to validate the structure. Both approaches are valid and can be used in the REST module:

**$I->sendGet('/users');**

**$I->seeResponseCodeIs(HttpCode::OK); *// 200***

**$I->seeResponseIsJson();**

**$I->seeResponseJsonMatchesJsonPath('$[0].user.login');**

**$I->seeResponseJsonMatchesXpath('//user/login');**

More detailed check can be applied if you need to validate the type of fields in a response. You can do that by using with a [seeResponseMatchesJsonType](https://codeception.com/docs/modules/REST#seeResponseMatchesJsonType) action in which you define the structure of JSON response.

**$I->sendGet('/users/1');**

**$I->seeResponseCodeIs(HttpCode::OK); *// 200***

**$I->seeResponseIsJson();**

**$I->seeResponseMatchesJsonType([**

**'id' => 'integer',**

**'name' => 'string',**

**'email' => 'string:email',**

**'homepage' => 'string:url|null',**

**'created\_at' => 'string:date',**

**'is\_active' => 'boolean'**

**]);**

Codeception uses this simple and lightweight definitions format which can be [easily learned and extended](https://codeception.com/docs/modules/REST#seeResponseMatchesJsonType).

### TAKING DATA FROM RESPONSES

When you need to obtain a value from a response and use it in next requests you can use grab\* methods. For instance, use grabDataFromResponseByJsonPath allows to query JSON for a value.

**list($id) = $I->grabDataFromResponseByJsonPath('$.id');**

**$I->sendGet('/pet/' . $id);**

### VALIDATING DATA JSON RESPONSES

The last line of the previous example verified that the response contained the provided string. However we shouldn’t rely on it, as depending on content formatting we can receive different results with the same data. What we actually need is to check that the response can be parsed and it contains some of the values we expect. In the case of JSON we can use the seeResponseContainsJson method

***// matches {"result":"ok"}'***

**$I->seeResponseContainsJson(['result' => 'ok']);**

***// it can match tree-like structures as well***

**$I->seeResponseContainsJson([**

**'user' => [**

**'name' => 'davert',**

**'email' => 'davert@codeception.com',**

**'status' => 'inactive'**

**]**

**]);**

You may want to perform even more complex assertions on a response. This can be done by writing your own methods in the [Helper](https://codeception.com/docs/06-ReusingTestCode#Modules-and-Helpers) classes. To access the latest JSON response you will need to get the response property of the REST module. Let’s demonstrate it with the seeResponseIsHtml method:

**namespace Helper;**

**class Api extends \Codeception\Module**

**{**

**public function seeResponseIsHtml()**

**{**

**$response = $this->getModule('REST')->response;**

**$this->assertRegExp('~^<!DOCTYPE HTML(.\*?)<html>.\*?<\/html>~m', $response);**

**}**

**}**

The same way you can receive request parameters and headers.

### TESTING XML RESPONSES

In case your REST API works with XML format you can use similar methods to test its data and structure. There is seeXmlResponseIncludes method to match inclusion of XML parts in response, and seeXmlResponseMatchesXpath to validate its structure.

**$I->sendGet('/users.xml');**

**$I->seeResponseCodeIs(\Codeception\Util\HttpCode::OK); *// 200***

**$I->seeResponseIsXml();**

**$I->seeXmlResponseMatchesXpath('//user/login');**

**$I->seeXmlResponseIncludes(\Codeception\Util\Xml::toXml([**

**'user' => [**

**'name' => 'davert',**

**'email' => 'davert@codeception.com',**

**'status' => 'inactive'**

**]**

**]));**

We are using Codeception\Util\Xml class which allows us to build XML structures in a clean manner. The toXml method may accept a string or array and returns \DOMDocument instance. If your XML contains attributes and so can’t be represented as a PHP array you can create XML using the [XmlBuilder](https://codeception.com/docs/reference/XmlBuilder) class. We will take a look at it a bit more in next section.

Use \Codeception\Util\Xml::build() to create XmlBuilder instance.

## SOAP API

SOAP web services are usually more complex. You will need PHP [configured with SOAP support](https://php.net/manual/en/soap.installation.php). Good knowledge of XML is required too. SOAP module uses specially formatted POST request to connect to WSDL web services. Codeception uses PhpBrowser or one of framework modules to perform interactions. If you choose using a framework module, SOAP will automatically connect to the underlying framework. That may improve the speed of a test execution and will provide you with more detailed stack traces.

Let’s configure SOAP module to be used with PhpBrowser:

**actor: ApiTester**

**modules:**

**enabled:**

**- SOAP:**

**depends: PhpBrowser**

**endpoint: http://serviceapp/api/v1/**

SOAP request may contain application specific information, like authentication or payment. This information is provided with SOAP header inside the <soap:Header> element of XML request. In case you need to submit such header, you can use haveSoapHeader action. For example, next line of code

**$I->haveSoapHeader('Auth', ['username' => 'Miles', 'password' => '123456']);**

will produce this XML header

**<soap:Header>**

**<Auth>**

**<username>Miles</username>**

**<password>123456</password>**

**</Auth>**

**</soap:Header>**

Use sendSoapRequest method to define the body of your request.

**$I->sendSoapRequest('CreateUser', '<name>Miles Davis</name><email>miles@davis.com</email>');**

This call will be translated to XML:

**<soap:Body>**

**<ns:CreateUser>**

**<name>Miles Davis</name>**

**<email>miles@davis.com</email>**

**</ns:CreateUser>**

**</soap:Body>**

And here is the list of sample assertions that can be used with SOAP.

**$I->seeSoapResponseEquals('<?xml version="1.0"<error>500</error>');**

**$I->seeSoapResponseIncludes('<result>1</result>');**

**$I->seeSoapResponseContainsStructure('<user><name></name><email></email>');**

**$I->seeSoapResponseContainsXPath('//result/user/name[@id=1]');**

In case you don’t want to write long XML strings, consider using [XmlBuilder](https://codeception.com/docs/reference/XmlBuilder) class. It will help you to build complex XMLs in jQuery-like style. In the next example we will use XmlBuilder instead of regular XML.

**$I->haveSoapHeader('Session', array('token' => '123456'));**

**$I->sendSoapRequest('CreateUser', Xml::build()**

**->user->email->val('miles@davis.com'));**

**$I->seeSoapResponseIncludes(\Codeception\Util\Xml::build()**

**->result->val('Ok')**

**->user->attr('id', 1)**

**);**

It’s up to you to decide whether to use XmlBuilder or plain XML. XmlBuilder will return XML string as well.

You may extend current functionality by using SOAP module in your helper class. To access the SOAP response as \DOMDocument you can use response property of SOAP module.

**namespace Helper;**

**class Api extends \Codeception\Module {**

**public function seeResponseIsValidOnSchema($schema)**

**{**

**$response = $this->getModule('SOAP')->response;**

**$this->assertTrue($response->schemaValidate($schema));**

**}**

**}**

## CONCLUSION

Codeception has two modules that will help you to test various web services. They need a new api suite to be created. Remember, you are not limited to test only response body. By including Db module you may check if a user has been created after the CreateUser call. You can improve testing scenarios by using REST or SOAP responses in your helper methods.

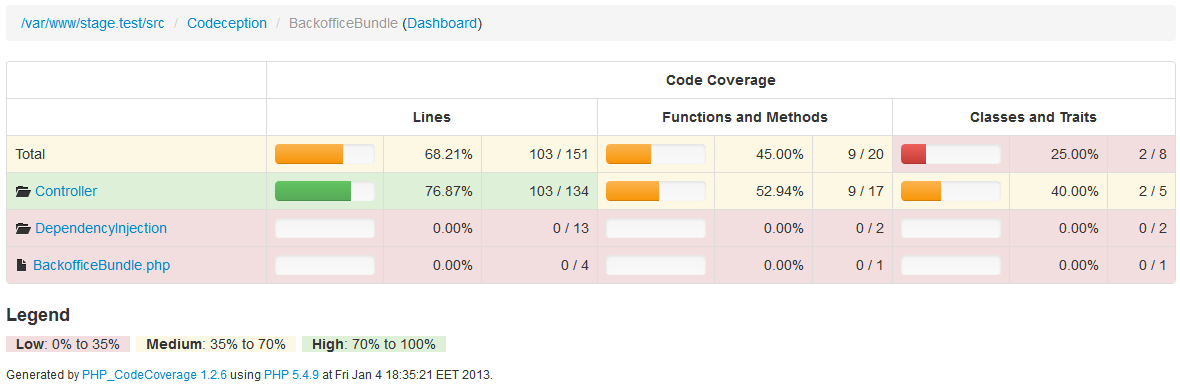
* **Next Chapter:**[**Codecoverage >**](https://codeception.com/docs/11-Codecoverage)
* **Previous Chapter:**[**< Data**](https://codeception.com/docs/09-Data)

# CODE COVERAGE

At some point you want to review which parts of your application are tested well and which are not. Just for this case the [CodeCoverage](https://en.wikipedia.org/wiki/Code_coverage) is used. When you execute your tests to collect coverage report, you will receive statistics of all classes, methods, and lines triggered by these tests. The ratio between all lines in script and all touched lines is a main coverage criterion. In the ideal world you should get 100% code coverage, but in reality 80% is really enough. Because even 100% code coverage rate doesn’t save you from fatal errors and crashes.

The required information is provided by [SebastianBergmann\CodeCoverage](https://github.com/sebastianbergmann/php-code-coverage), and you can use any of the supported drivers.

| **Driver** | **Description** |
| --- | --- |
| [Xdebug](https://github.com/xdebug/xdebug) | Great for debugging, but too slow when collecting coverage |
| [phpdbg](https://www.php.net/manual/en/book.phpdbg.php) | Faster than Xdebug but inaccurate |
| [pcov](https://github.com/krakjoe/pcov) | Fast and accurate, but no debug functionality — perfect for CI |



Coverage data can be collected manually for both local and remote tests. Remote tests may be executed on different nodes, or locally but running through web server. It may look hard to collect code coverage for Selenium tests or PhpBrowser tests. But Codeception supports remote codecoverage as well as local.

### CONFIGURATION

To enable code coverage put these lines in the global configuration file codeception.yml:

**coverage:**

**enabled: true**

That’s ok for now. But what files should be present in final coverage report? Pass an array of files or directory to include/exclude sections. The path ending with ‘\*’ matches the directory. Also you can use ‘\*’ mask in a file name, i.e. app/models/\*Model.php to match all models.

There is a shortcut if you don’t need that complex filters:

**coverage:**

**enabled: true**

**include:**

**- app/\***

**exclude:**

**- app/cache/\***

Include and exclude options can be redefined for each suite in corresponding config files.

By default, if coverage is reported to be < 35% it is marked as low, and >70% is high coverage. You can also define high and low boundaries with low\_limit and high\_limit config options:

**coverage:**

**enabled: true**

**low\_limit: 30**

**high\_limit: 60**

By default, show all whitelisted files in --coverage-text output not just the ones with coverage information is set to false, config option:

**coverage:**

**enabled: true**

**show\_uncovered: false**

By default, show only the coverage report summary in --coverage-text output is set to false, config option:

**coverage:**

**enabled: true**

**show\_only\_summary: false**

For further information please refer to the [PHPUnit configuration docs](https://phpunit.readthedocs.io/en/latest/configuration.html)

## LOCAL CODECOVERAGE

The basic codecoverage can be collected for functional and unit tests. If you performed configuration steps from above, you are ready to go. All you need is to execute codeception with --coverage option.

To generate a clover xml report or a tasty html report append also --coverage-xml and --coverage-html options.

**codecept run --coverage --coverage-xml --coverage-html**

XML and HTML reports are stored to the \_output directory. The best way to review report is to open index.html from tests/\_output/coverage in your browser. XML clover reports are used by IDEs (like PHPStorm) or Continuous Integration servers (like Jenkins).

## REMOTE CODECOVERAGE

### LOCAL SERVER

If you run your application via Webserver (Apache, Nginx, PHP WebServer) you don’t have a direct access to tested code, so collecting coverage becomes a non-trivial task. The same goes for scripts that are tested on different nodes. To get access to this code you need xdebug installed with remote\_enable option turned on. Codeception also requires a little spy to interact with your application. As your application runs standalone, without even knowing it is being tested, a small file should be included in order to collect coverage info.

This file is called c3.php and is [available on GitHub](https://github.com/Codeception/c3). c3.php should be downloaded and included in your application at the very first line from controller. By sending special headers Codeception will command your application when to start codecoverage collection and when to stop it. After the suite is finished, a report will be stored and Codeception will grab it from your application.

Please, follow installation instructions described in a [readme file](https://github.com/Codeception/c3).

To connect to c3 Codeception uses url config from PhpBrowser or WebDriver module. But URL of index with c3.php included can be specified explicitly with c3\_url parameter defined:

**coverage:**

***# url of file which includes c3 router.***

**c3\_url: 'http://127.0.0.1:8000/index-test.php/'**

note: we can’t have multiple c3\_url on same host difference only by port. Please, use alias of domain (i.e. frontend.dev:8000,backend.dev:8080) instead.

After the c3.php file is included in application you can start gather coverage. In case you execute your application locally there is nothing to be changed in config. All codecoverage reports will be collected as usual and merged afterwards. Think of it: Codeception runs remote coverage in the same way as local.

#### Custom cookie domain

It’s possible to override the cookie domain set by Codeception during code coverage. Typical case for that is when you have several subdomains that your acceptance tests are visiting, e.g. mysite.com and admin.mysite.com. By default, Codeception will run code coverage only for the domain set in the url of the WebDriver/url (or c3\_url if defined), thus leaving out other subdomains from code coverage. To avoid that and to include all relevant subdomains in code covereage, it’s advised to set .mysite.com as the cookie domain option:

**coverage:**

**cookie\_domain: ".mysite.com"**

### REMOTE SERVER

But if you run tests on different server (or your webserver doesn’t use code from current directory) a single option remote should be added to config. For example, let’s turn on remote coverage for acceptance suite in acceptance.suite.yml:

**coverage:**

**remote: true**

In this case remote Code Coverage results won’t be merged with local ones, if this option is enabled. Merging is possible only in case a remote and local files have the same path. But in case of running tests on a remote server we are not sure of it.

CodeCoverage results from remote server will be saved to tests/\_output directory. Please note that remote codecoverage results won’t be displayed in console by the reason mentioned above: local and remote results can’t be merged, and console displays results for local codecoverage.

### WORKING DIRECTORY (DOCKER/SHARED MOUNTS)

If your remote server is accessed through a shared mount, or a mounted folder (IE: Docker Volumes), you can still get merged coverage details. Use the work\_dir option to specify the work directory. When CodeCoverage runs, Codeception will update any path that matches the work\_dir option to match the local current project directory.

Given a docker command similar to:

**docker run -v *$(*pwd*)*:/workdir -w /workdir...**

Use the below configuration to allow coverage mergers.

**coverage:**

**remote: false**

**work\_dir: /workdir**

### REMOTE CONTEXT OPTIONS

HTML report generation can at times take a little more time than the default 30 second timeout. Or maybe you want to alter SSL settings (verify\_peer, for example) To alter the way c3 sends it’s service requests to your webserver (be it a local or a remote one), you can use the remote\_context\_options key in coverage settings.

**coverage:**

**remote\_context\_options:**

**http:**

**timeout: 60**

**ssl:**

**verify\_peer: false**

Context stream options are [well documented at php.net](https://php.net/manual/en/context.php)

## CONCLUSION

It’s never been easier to setup local and remote code coverage. Just one config and one additional file to include! **With Codeception you can easily generate CodeCoverage reports for your Selenium tests** (or other acceptance or api tests). Mixing reports for acceptance, functional, and unit suites provides you with the most complete information on which parts of your applications are tested and which are not.

* **Next Chapter:**[**ContinuousIntegration >**](https://codeception.com/docs/12-ContinuousIntegration)
* **Previous Chapter:**[**< APITesting**](https://codeception.com/docs/10-APITesting)

# CONTINUOUS INTEGRATION

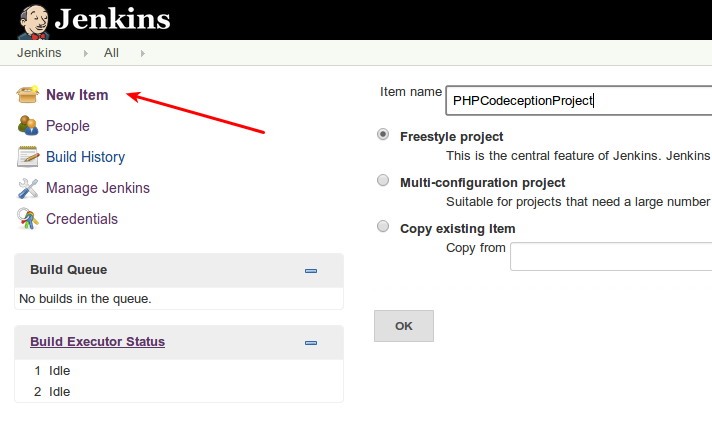
Once you get testing suite up and running you are interested in running your tests regularly. If you ensure that tests are running on every code change or at least once a day you can be sure that no regression is introduced. This allows to keep you system stable. But developers are not so passionate about running all tests manually, they also can forget to execute tests before pushing code to production… The solution is simple, test execution should be automated. Instead of running them locally it is better to have dedicated server responsible for running tests for a team. This way we can ensure that everyone’s tests executed, which commit made a regression in codebase, and that we can deploy only once tests pass.

There are many Continuous Integration Servers out there. We will try to list basic steps to setup Codeception tests with them. If your CI system is not mentioned, you can get the idea by analogy. Please also help us to extend this guide by adding instructions for different CIs.

## JENKINS



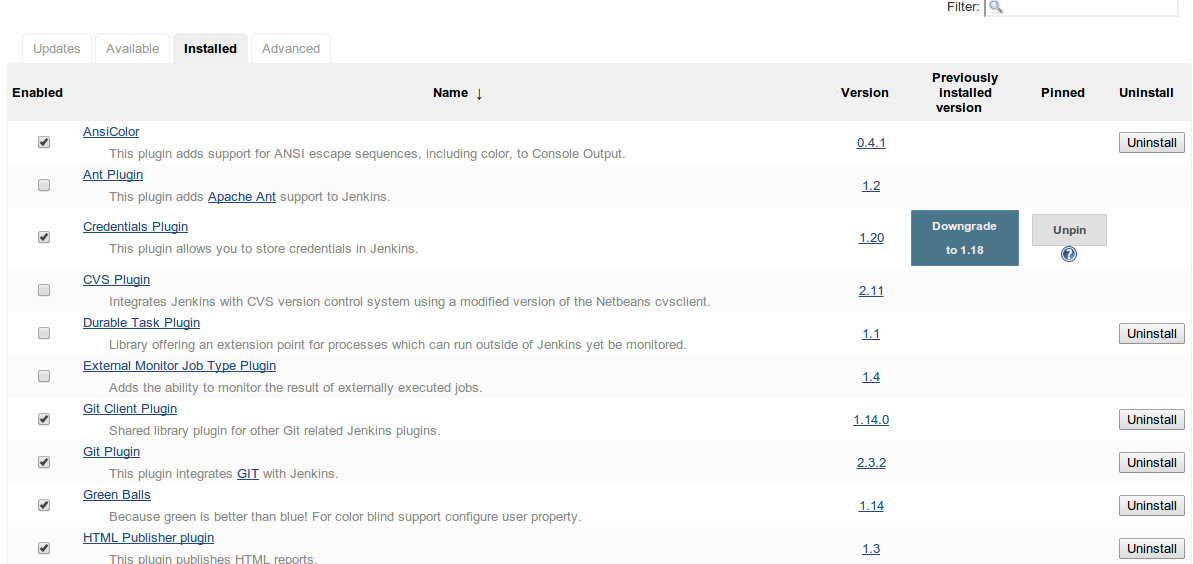
[Jenkins](https://jenkins-ci.org/) is one of the most popular open-source solution on market. It is easy to setup and is easy to customize by applying various plugins.



### PREPARING JENKINS

It is recommended to have the next plugins installed:

* **Git Plugin** - for building tests for Git repo
* **Green Balls** - to display success results in green.
* **xUnit Plugin**, **jUnit Plugin** - to process and display Codeception XML reports
* **HTML Publisher Plugin** - to process Codeception HTML reports
* **AnsiColor** - to show colorized console output.



### BASIC SETUP

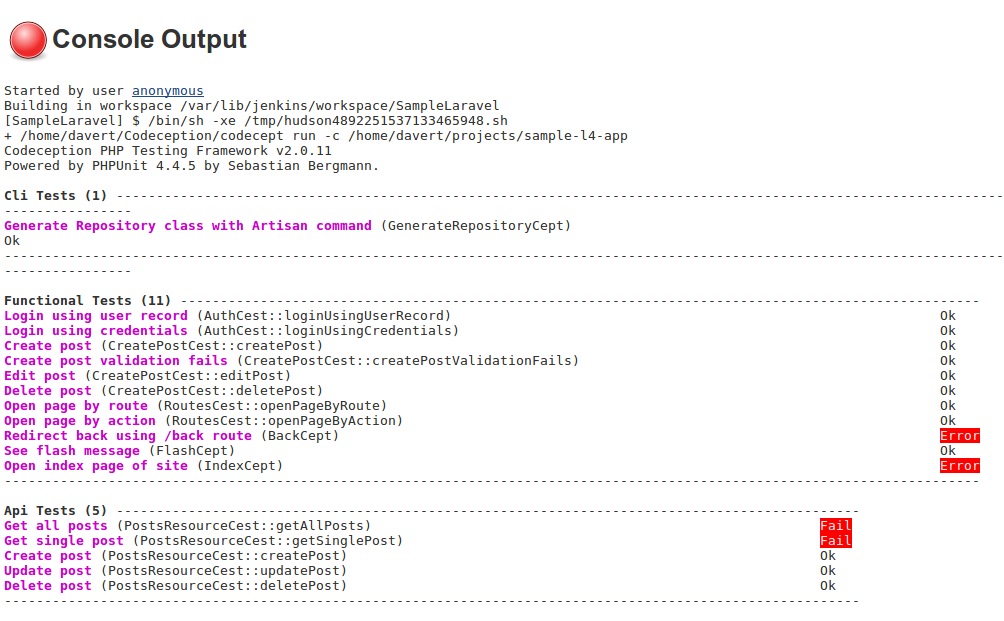
At first we need to create build project. Depending on your needs you can set up periodical build or trigger build once the change is pushed to GitHub (you will need GitHub plugin for that).

We need to define build steps. The most simple setup may look like this:

**vendor/bin/codecept run**



Then we can start the very first job and check the execution progress. If tests fail we will see that in console:



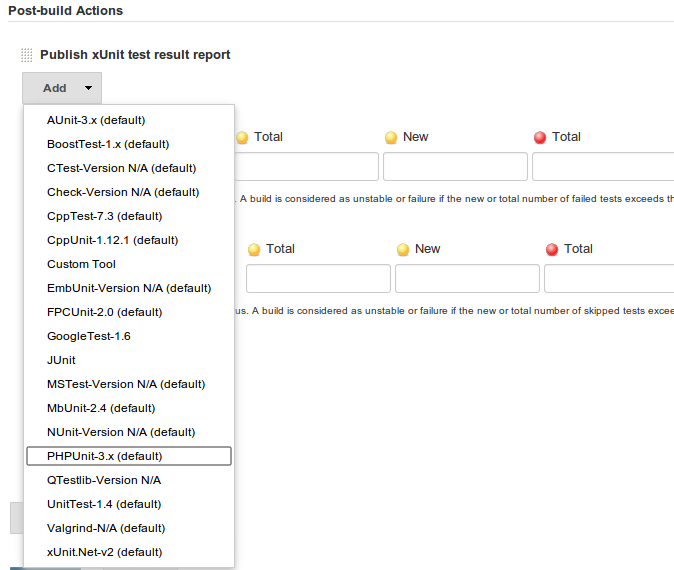
### XML REPORTS

But we don’t want to analyze console output for each failing build. Especially If Jenkins can collect and display the results inside its web UI. Codeception can export its results using JUnit XML format. To generate XML report on each build we will need to append --xml option to Codeception execution command. Codeception will print result.xml file containing information about test status with steps and stack traces for failing tests.

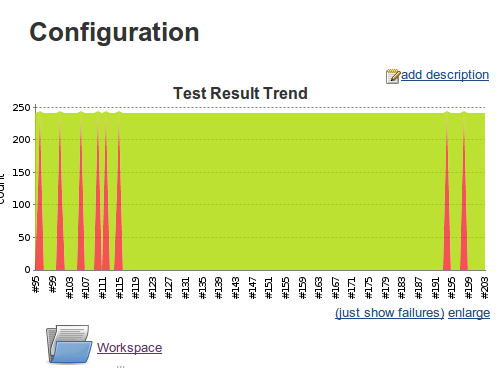
Now let’s update our build step to generate xml:

**vendor/bin/codecept run --xml**

and ask Jenkins to collect resulted XML. This can be done as part of Post-build actions. Let’s add Publish xUnit test result report action and configure it to use with PHPUnit reports.



Now we should specify path to PHPUnit style XML reports. In case of standard Codeception setup we should specify tests/\_output/\*.xml as a pattern for matching resulted XMLs. Now we save the project and rebuild it.



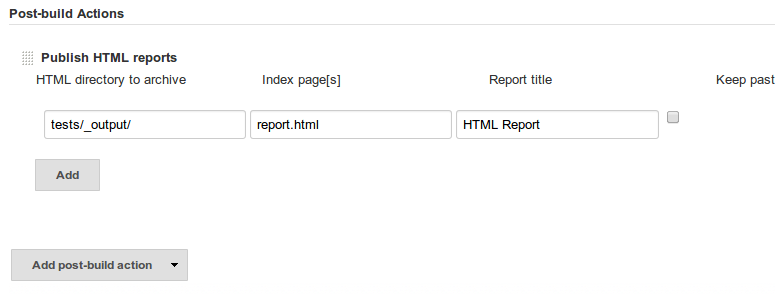
Now for all builds we will see results trend graph that shows us percentage of passing and failing tests. We also will see a **Latest Test Result** link which will lead to to the page where all executed tests and their stats listed in a table.

### HTML REPORTS

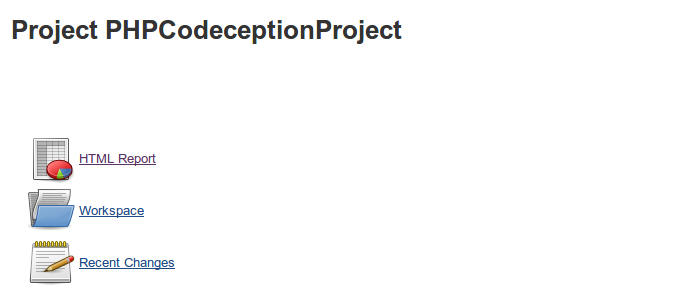
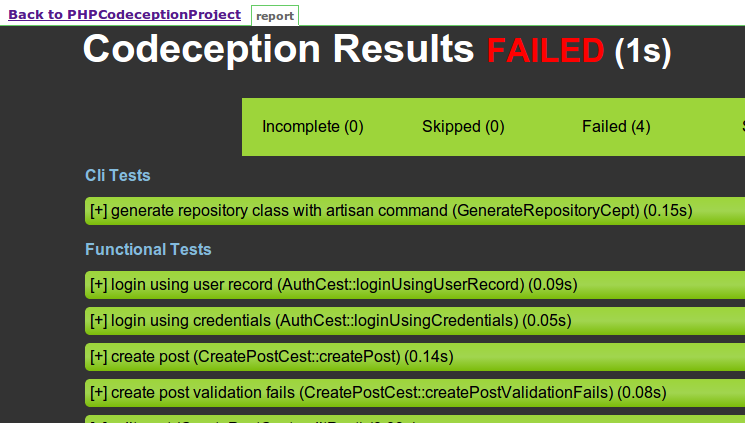
To get more details on steps executed you can generate HTML report and use Jenkins to display them.

**vendor/bin/codecept run --html**

Now we need HTML Publisher plugin configured to display generated HTML files. It should be added as post-build action similar way we did it for XML reports.



Jenkins should locate report.html at tests/\_output/. Now Jenkins will display HTML reports for each build.

## TEAMCITY



TeamCity is a hosted solution from JetBrains. The setup of it can be a bit tricky as TeamCity uses its own reporter format for parsing test results. PHPUnit since version 5.x has integrated support for this format, so does Codeception. What we need to do is to configure Codeception to use custom reporter. By default there is --report option which provides an alternative output. You can change the reporter class in codeception.yml configuration:

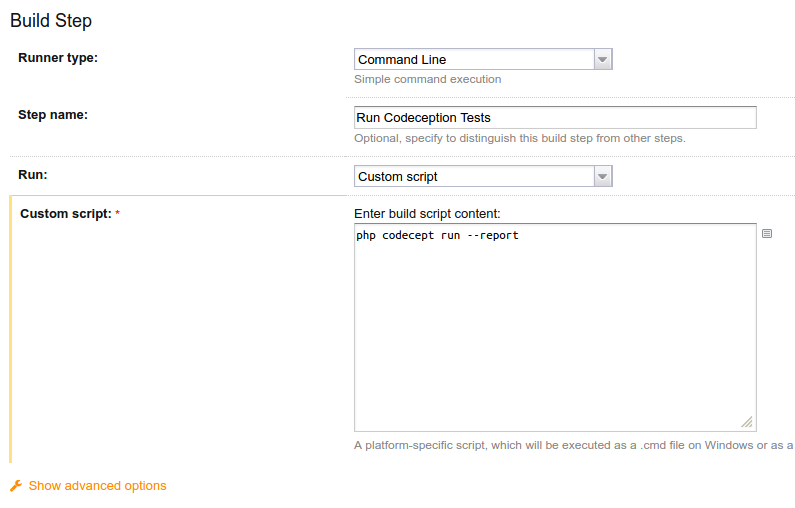
**reporters:**

**report: PHPUnit\_Util\_Log\_TeamCity**

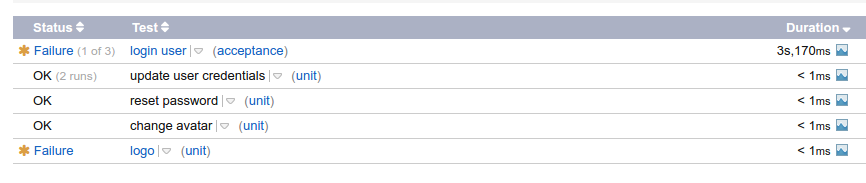
As an alternative you can use 3rd-party [TeamCity extension](https://github.com/neronmoon/TeamcityCodeception) for better reporting.

After you create build project you should define build step with Codeception which is

**vendor/bin/codecept run --report**



Once you execute your first build you should see detailed report inside TeamCity interface:



## TRAVISCI



Travis CI is popular service CI with good GitHub integration. Codeception is self-tested with Travis CI. There nothing special about configuration. Just add to the bottom line of travis configuration:

**php vendor/bin/codecept run**

More details on configuration can be learned from Codeception’s [.travis.yml](https://github.com/Codeception/Codeception/blob/3.0/.travis.yml).

Travis doesn’t provide visualization for XML or HTML reports so you can’t view reports in format any different than console output. However, Codeception produces nice console output with detailed error reports.

## GITLAB



If a file .gitlab-ci.yml exists in the root of the git repository, GitLab will run a pipeline each time you push to the gitlab server. The file configures the docker image that will be called. Below is a sample which loads a php7 docker image, clones your files, installs composer dependencies, runs the built-in php webserver and finally runs codeception:

***# Select image from https://hub.docker.com/\_/php/***

**image: php:7.0**

***# Select what we should cache***

**cache:**

**paths:**

**- vendor/**

**before\_script:**

***# Install git and unzip (composer will need them)***

**- apt-get update && apt-get install -qqy git unzip**

***# Install composer***

**- curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer**

***# Install all project dependencies***

**- composer install**

***# Run webserver***

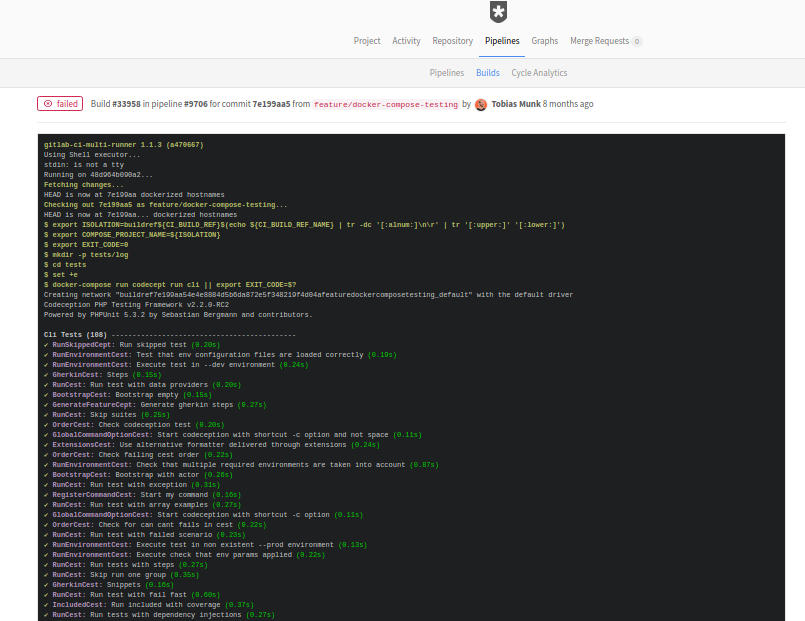
**- php -S localhost:8085 --docroot public &>/dev/null&**

***# Test***

**test:**

**script:**

**- vendor/bin/codecept run**



For acceptance testing you can use codeception/codeception docker image as base. See example below:

**image:**

**name: codeception/codeception**

***# clear image entrypoint to make bash being available***

**entrypoint: [""]**

***# run selenium chrome as a local service (put "host: 'selenium\_\_standalone-chrome'" in environment configuration)***

**services:**

**- selenium/standalone-chrome:latest**

***# Select what we should cache***

**cache:**

**paths:**

**- vendor/**

**before\_script:**

***# Install all project dependencies***

**- composer install**

***# Test***

**test:**

**script:**

**- vendor/bin/codecept run acceptance --xml --html**

**artifacts:**

**when: always**

**expire\_in: 1 week**

**paths:**

**- tests/\_output**

***# make the report available in Gitlab UI. see https://docs.gitlab.com/ee/ci/unit\_test\_reports.html***

**reports:**

**junit: tests/\_output/report.xml**

## CONCLUSION

It is highly recommended to use Continuous Integration system in development. Codeception is easy to install and run in any CI systems. However, each of them has their differences you should take into account. You can use different reporters to provide output in format expected by CI system.

* **Next Chapter:**[**ParallelExecution >**](https://codeception.com/docs/12-ParallelExecution)
* **Previous Chapter:**[**< Codecoverage**](https://codeception.com/docs/11-Codecoverage)

# PARALLEL EXECUTION

When execution time of your tests is longer than a coffee break, it is a good reason to think about making your tests faster. If you have already tried to run them on SSD drive, and the execution time still upsets you, it might be a good idea to run your tests in parallel.

## WHERE TO START

Codeception does not provide a command like run-parallel. There is no common solution that can play well for everyone. Here are the questions you will need to answer:

* How parallel processes will be executed?
* How parallel processes won’t affect each other?
* Will they use different databases?
* Will they use different hosts?
* How should I split my tests across parallel processes?

There are two approaches to achieve parallelization. We can use [Docker](https://docker.com/) and run each process inside isolated containers, and have those containers executed simultaneously.

Docker works really well for isolating testing environments. By the time of writing this chapter, we didn’t have an awesome tool like it. This chapter demonstrates how to manage parallel execution manually. As you will see we spend too much effort trying to isolate tests which Docker does for free. Today we **recommend using Docker** for parallel testing.

## DOCKER

Please make sure you have docker or [Docker Toolbox](https://www.docker.com/products/docker-toolbox) installed. Docker experience is required as well.

### USING CODECEPTION DOCKER IMAGE

Run official Codeception image from DockerHub:

**docker run codeception/codeception**

Running tests from a project, by mounting the current path as a host-volume into the container. The **default working directory in the container is /project**.

**docker run -v ${PWD}:/project codeception/codeception run**

To prepare application and tests to be executed inside containers you will need to use [Docker Compose](https://docs.docker.com/compose/) to run multiple containers and connect them together.

Define all required services in docker-compose.yml file. Make sure to follow Docker philisophy: 1 service = 1 container. So each process should be defined as its own service. Those services can use official Docker images pulled from DockerHub. Directories with code and tests should be mounted using volume directive. And exposed ports should be explicitly set using ports directive.

We prepared a sample config with codeception, web server, database, and selenium with Chrome to be executed together.

**version: '3'**

**services:**

**codecept:**

**image: codeception/codeception**

**depends\_on:**

**- chrome**

**- web**

**volumes:**

**- .:/project**

**web:**

**image: php:7-apache**

**depends\_on:**

**- db**

**volumes:**

**- .:/var/www/html**

**db:**

**image: percona:5.6**

**chrome:**

**image: selenium/standalone-chrome**

Codeception service will execute command codecept run but only after all services are started. This is defined using depends\_on parameter.

It is easy to add more custom services. For instance to use Redis you just simple add this lines:

**redis:**

**image: redis:3**

By default the image has codecept as its entrypoint, to run the tests simply supply the run command

**docker-compose run --rm codecept help**

Run suite

**docker-compose run --rm codecept run acceptance**

**docker-compose run --rm codecept run acceptance LoginCest**

Development bash

**docker-compose run --rm --entrypoint bash codecept**

And finally to execute testing in parallel you should define how you split your tests and run parallel processes for docker-compose. Here we split tests by suites, but you can use different groups to split your tests. In section below you will learn how to do that with Robo.

**docker-compose --project-name test-web run -d --rm codecept run --html report-web.html web & \**

**docker-compose --project-name test-unit run -d --rm codecept run --html report-unit.html unit & \**

**docker-compose --project-name test-functional run -d --rm codecept run --html report-functional.html functional**

At the end, it is worth specifying that Docker setup can be complicated and please make sure you understand Docker and Docker Compose before proceed. We prepared some links that might help you:

* [Acceptance Tests Demo Repository](https://github.com/dmstr/docker-acception)
* [Dockerized Codeception Internal Tests](https://github.com/Codeception/Codeception/blob/master/tests/README.md#dockerized-testing)
* [Phundament App with Codeception](https://gist.github.com/schmunk42/d6893a64963509ff93daea80f722f694)

If you want to automate splitting tests by parallel processes, and executing them using PHP script you should use Robo task runner to do that.

## ROBO

### WHAT TO DO

Parallel Test Execution consists of 3 steps:

* splitting tests
* running tests in parallel
* merging results

We propose to perform those steps using a task runner. In this guide we will use [**Robo**](http://robo.li/) task runner. It is a modern PHP task runner that is very easy to use. It uses [Symfony Process](https://symfony.com/doc/current/components/process.html) to spawn background and parallel processes. Just what we need for the step 2! What about steps 1 and 3? We have created robo [tasks](https://github.com/Codeception/robo-paracept) for splitting tests into groups and merging resulting JUnit XML reports.

To conclude, we need:

* [Robo](http://robo.li/), a task runner.
* [robo-paracept](https://github.com/Codeception/robo-paracept) - Codeception tasks for parallel execution.

## PREPARING ROBO AND ROBO-PARACEPT

Execute this command in an empty folder to install Robo and Robo-paracept :

**$ composer require codeception/robo-paracept:dev-master**

You need to install Codeception after, if codeception is already installed it will not work.

**$ composer require codeception/codeception**

### PREPARING ROBO

Initializes basic RoboFile in the root of your project

**$ robo init**

Open RoboFile.php to edit it

**class RoboFile extends \Robo\Tasks**

**{**

***// define public methods as commands***

**}**

Each public method in robofile can be executed as a command from console. Let’s define commands for 3 steps and include autoload.

**require\_once 'vendor/autoload.php';**

**class Robofile extends \Robo\Tasks**

**{**

**use \Codeception\Task\MergeReports;**

**use \Codeception\Task\SplitTestsByGroups;**

**public function parallelSplitTests()**

**{**

**}**

**public function parallelRun()**

**{**

**}**

**public function parallelMergeResults()**

**{**

**}**

**}**

If you run robo, you can see the respective commands:

**$ robo**

**Robo version 0.6.0**

**Usage:**

**command [options] [arguments]**

**Options:**

**-h, --help Display this help message**

**-q, --quiet Do not output any message**

**-V, --version Display this application version**

**--ansi Force ANSI output**

**--no-ansi Disable ANSI output**

**-n, --no-interaction Do not ask any interactive question**

**-v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug**

**Available commands:**

**help Displays help for a command**

**list Lists commands**

**parallel**

**parallel:merge-results**

**parallel:run**

**parallel:split-tests**

#### Step 1: Split Tests

Codeception can organize tests into [groups](https://codeception.com/docs/07-AdvancedUsage#Groups). Starting from 2.0 it can load information about a group from a files. Sample text file with a list of file names can be treated as a dynamically configured group. Take a look into sample group file:

**tests/functional/LoginCept.php**

**tests/functional/AdminCest.php:createUser**

**tests/functional/AdminCest.php:deleteUser**

Tasks from \Codeception\Task\SplitTestsByGroups will generate non-intersecting group files. You can either split your tests by files or by single tests:

**function parallelSplitTests()**

**{**

***// Split your tests by files***

**$this->taskSplitTestFilesByGroups(5)**

**->projectRoot('.')**

**->testsFrom('tests/acceptance')**

**->groupsTo('tests/\_data/paracept\_')**

**->run();**

***/\****

***// Split your tests by single tests (alternatively)***

***$this->taskSplitTestsByGroups(5)***

***->projectRoot('.')***

***->testsFrom('tests/acceptance')***

***->groupsTo('tests/\_data/paracept\_')***

***->run();***

***\*/***

**}**

Let’s prepare group files:

**$ robo parallel:split-tests**

**[Codeception\Task\SplitTestFilesByGroupsTask] Processing 33 files**

**[Codeception\Task\SplitTestFilesByGroupsTask] Writing tests/\_data/paracept\_1**

**[Codeception\Task\SplitTestFilesByGroupsTask] Writing tests/\_data/paracept\_2**

**[Codeception\Task\SplitTestFilesByGroupsTask] Writing tests/\_data/paracept\_3**

**[Codeception\Task\SplitTestFilesByGroupsTask] Writing tests/\_data/paracept\_4**

**[Codeception\Task\SplitTestFilesByGroupsTask] Writing tests/\_data/paracept\_5**

Now we have group files. We should update codeception.yml to load generated group files. In our case we have groups: paracept\_1, paracept\_2, paracept\_3, paracept\_4, paracept\_5.

**groups:**

**paracept\_\*: tests/\_data/paracept\_\***

Let’s try to execute tests from the second group:

**$ codecept run acceptance -g paracept\_2**

#### Step 2: Running Tests

Robo has ParallelExec task to spawn background processes.

##### Inside Container

If you are using [Docker](https://codeception.com/docs/12-ParallelExecution#docker) containers you can launch multiple Codeception containers for different groups:

**public function parallelRun()**

**{**

**$parallel = $this->taskParallelExec();**

**for ($i = 1; $i <= 5; $i++) {**

**$parallel->process(**

**$this->taskExec('docker-compose run --rm codecept run')**

**->option('group', "paracept\_$i") *// run for groups paracept\_\****

**->option('xml', "tests/\_log/result\_$i.xml") *// provide xml report***

**);**

**}**

**return $parallel->run();**

**}**

##### Locally

If you want to run tests locally just use preinstalled taskCodecept task of Robo to define Codeception commands and put them inside parallelExec.

**public function parallelRun()**

**{**

**$parallel = $this->taskParallelExec();**

**for ($i = 1; $i <= 5; $i++) {**

**$parallel->process(**

**$this->taskCodecept() *// use built-in Codecept task***

**->suite('acceptance') *// run acceptance tests***

**->group("paracept\_$i") *// for all paracept\_\* groups***

**->xml("tests/\_log/result\_$i.xml") *// save XML results***

**);**

**}**

**return $parallel->run();**

**}**

In case you don’t use containers you can isolate processes by starting different web servers and databases per each test process.

We can define different databases for different processes. This can be done using [Environments](https://codeception.com/docs/07-AdvancedUsage#Environments). Let’s define 5 new environments in acceptance.suite.yml:

**actor: AcceptanceTester**

**modules:**

**enabled:**

**- Db:**

**dsn: 'mysql:dbname=testdb;host=127.0.0.1'**

**user: 'root'**

**dump: 'tests/\_data/dump.sql'**

**populate: true**

**cleanup: true**

**- WebDriver:**

**url: 'http://localhost/'**

**env:**

**env1:**

**modules:**

**config:**

**Db:**

**dsn: 'mysql:dbname=testdb\_1;host=127.0.0.1'**

**WebDriver:**

**url: 'http://test1.localhost/'**

**env2:**

**modules:**

**config:**

**Db:**

**dsn: 'mysql:dbname=testdb\_2;host=127.0.0.1'**

**WebDriver:**

**url: 'http://test2.localhost/'**

**env3:**

**modules:**

**config:**

**Db:**

**dsn: 'mysql:dbname=testdb\_3;host=127.0.0.1'**

**WebDriver:**

**url: 'http://test3.localhost/'**

**env4:**

**modules:**

**config:**

**Db:**

**dsn: 'mysql:dbname=testdb\_4;host=127.0.0.1'**

**WebDriver:**

**url: 'http://test4.localhost/'**

**env5:**

**modules:**

**config:**

**Db:**

**dsn: 'mysql:dbname=testdb\_5;host=127.0.0.1'**

**WebDriver:**

**url: 'http://test5.localhost/'**

After the parallelRun method is defined you can execute tests with

**$ robo parallel:run**

#### Step 3: Merge Results

In case of parallelExec task we recommend to save results as JUnit XML, which can be merged and plugged into Continuous Integration server.

**function parallelMergeResults()**

**{**

**$merge = $this->taskMergeXmlReports();**

**for ($i=1; $i<=5; $i++) {**

**$merge->from("tests/\_output/result\_paracept\_$i.xml");**

**}**

**$merge->into("tests/\_output/result\_paracept.xml")->run();**

**}**

Now, we can execute :

**$ robo parallel:merge-results**

result\_paracept.xml file will be generated. It can be processed and analyzed.

#### All Together

To create one command to rule them all we can define new public method parallelAll and execute all commands. We will save the result of parallelRun and use it for our final exit code:

**function parallelAll()**

**{**

**$this->parallelSplitTests();**

**$result = $this->parallelRun();**

**$this->parallelMergeResults();**

**return $result;**

**}**

## CONCLUSION

Codeception does not provide tools for parallel test execution. This is a complex task and solutions may vary depending on a project. We use [Robo](http://robo.li/) task runner as an external tool to perform all required steps. To prepare our tests to be executed in parallel we use Codeception features of dynamic groups and environments. To do even more we can create Extensions and Group classes to perform dynamic configuration depending on a test process.

* **Previous Chapter:**[**< ContinuousIntegration**](https://codeception.com/docs/12-ContinuousIntegration)