live search...

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:

```
<?php
$I = new FunctionalTester($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 use the `codecept build` command. It generates method signatures from

Follow @codeception

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 confli

Follow @codeception

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.

```
<?php
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:

```
<?php
$I->seePageReloaded();
$I->seeClassIsLoaded($classname);
$I->dontSeeUserExist($user);
```

And then use them in your tests:

Follow @codeception

```php
<?php
$I->seePageReloaded();
$I->seeClassIsLoaded('FunctionalTester');
$I->dontSeeUserExist($user);
```

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

```php
<?php

function seeClassExist($class)
{
    $this->assertTrue(class_exists($class));
}
```

In your helpers you can use these assertions:

```php
<?php

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.

```php
<?php

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.

Follow @codeception

```php
<?php
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:

```php
<?php
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 (http://codeception.com/docs/reference/Commands) and are listed here. You are free to redefine them in your module.

Follow @codeception

```php
<?php

    // 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:

Follow @codeception

```
<?php
    $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:

```
<?php
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.

```
<?php
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.

Follow @codeception

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):

```
module:
    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 can use the `_reconfigure` method of the module. You may call it from a helper class and pass in all the fields you want to change.

In this case configuration will be changed instantly. In next example we change root URL for PhpBrowser to point to the admin area, so next `amOnPage('/')` will open `/admin/` page.

```
<?php
$this->getModule('PhpBrowser')->_reconfigure(array('url' => 'http://localhost/admin'));
```

Follow @codeception

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 `_restart` method which takes an array with config and restarts the browser.

```
<?php
// 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 (http://codeception.com/docs/07-AdvancedUsage#Cest-Classes) and Test\Unit (http://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 (http://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`.

```
<?php
/**
 * @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 (http://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 > (/docs/06-ReusingTestCode)**
- **Previous Chapter: < UnitTests (/docs/05-UnitTests)**

Follow @codeception

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

Thanks to

(https://www.jetbrains.com/phpstorm/) (https://www.rebilly.com/)

(https://github.com/codeception/codeception) (https://twitter.com/codeception)

(http://www.facebook.com/pages/Codeception/288959711204412)

- Installation (/install)
- Credits (/credits)
- Releases (/changelog)
- Commercial Services (http://sdclabs.com/codeception?utm_source=codeception.com&utm_medium=bottom_menu&utm_term=link&utm_campaign=reference)
- License (https://github.com/Codeception/Codeception/blob/master/LICENSE)

Codeception Family

Robo

(http://robo.li)(http://robo.li)
Modern PHP **Task Runner**. Allows to declare tasks with zero configuration in pure PHP.

CodeceptJS

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

Follow @codeception