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.

In simple terms we set the `$_REQUEST`, `$_GET` and `$_POST` variables and then we execute the application from a test. This may be valuable, as functional tests are faster and provide detailed stack traces on failures.

Codeception can connect to different PHP frameworks that support functional testing: Symfony2, Laravel5, Yii2, Zend Framework and others. You just need to enable the desired module in your functional suite configuration to start.

Modules for all of these frameworks share the same interface, and thus your tests are not bound to any one of them. This is a sample functional test:

```
<?php
// 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 Symfony2
    }
}
```

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 URL's

Functional tests cannot access external URL's, just URL's within your project. You can use Guzzle to open external URL's.

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
        - \Helper\Functional
```

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 (http://codeception.com/docs/modules/Symfony)

Laravel5

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

```
# functional.suite.yml

actor: FunctionalTester
modules:
    enabled:
        - Laravel5
        - \Helper\Functional
```

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.

Yii

By itself Yii framework does not have an engine for functional testing. So Codeception is the first and the only functional testing framework for Yii. To use it with Yii include `Yii1` module into config:

```
# functional.suite.yml

actor: FunctionalTester
modules:
    enabled:
        - Yii1
        - \Helper\Functional
```

To avoid the common pitfalls we discussed earlier, Codeception provides basic hooks over the Yii engine. Please set them up following the installation steps in the module reference (http://codeception.com/docs/modules/Yii1).

Zend Framework 2

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

```
# functional.suite.yml

actor: FunctionalTester
modules:
    enabled:
        - ZF2
        - \Helper\Functional
```

Zend Framework 1.x

The module for Zend Framework is highly inspired by the ControllerTestCase class, used for functional testing with PHPUnit. It follows similar approaches for bootstrapping and cleaning up. To start using Zend Framework in your functional tests, include the `ZF1` module:

```
# functional.suite.yml

actor: FunctionalTester
modules:
    enabled:
        - ZF1
        - \Helper\Functional
```

See the full reference (http://codeception.com/docs/modules/ZF1)

Phalcon

The `Phalcon` 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 `Phalcon` module and provide the path to this bootstrap file:

```
# functional.suite.yml

actor: FunctionalTester
modules:
    enabled:
        - Phalcon:
            bootstrap: 'app/config/bootstrap.php'
             cleanup: true
             savepoints: true
        - \Helper\Functional
```

See the full reference (http://codeception.com/docs/modules/Phalcon)

Writing Functional Tests

Functional tests are written in the same manner as Acceptance Tests (http://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:

```
<?php
$I = new FunctionalTester($scenario);
$I->amOnPage('/login');
```

We can click links to open web pages:

```
<?php
$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:

```
<?php
$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:

```
<?php
$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 exist