**Функциональные тесты - кратко**

У Codeception нет базового модуля для функционального тестирования. Поэтому в конфигурации необходимо включить один из поддерживаемых модулей фреймворков. Нужно это для того, что бы при запуске тестов не использовать веб сервер.

С полным списком модулей можно ознакомится на [GitHub](https://github.com/Codeception/Codeception/tree/master/docs/modules) или [официальном сайте](http://codeception.com/). Если в списке нет подходящего модуля можно интегрировать свой фреймворк. Делается это созданием моста между [BrowserKit](https://github.com/symfony/browser-kit) (его использует Codeception) и нашим приложением.

Мы будем использовать собственный модуль Gateway. Этот модуль подойдет если наш фреймворк использует Symfony HttpKernel.

Сначала создаем сам модуль по адресу tests/modules/Gateway.php:

namespace Codeception\Module;

use Codeception\TestCase;

use Symfony\Component\HttpKernel\Client;

class Gateway extends \Codeception\Lib\Framework

{

public function \_initialize()

{

require dirname(dirname(\_\_DIR\_\_)) . '/vendor/autoload.php';

$config = require dirname(\_\_DIR\_\_) . '/config/main.php';

$this->kernel = new \app\framework\Frontend($config);

}

public function \_before(TestCase $test)

{

$this->client = new Client($this->kernel);

$this->client->followRedirects(true);

}

public function isStatusCode($code)

{

return $this->kernel->handle()->getStatusCode() === $code;

}

}

Затем подключаем его в файле конфигурации (tests/functional.suite.yml):

class\_name: FunctionalTester

modules:

enabled: [Gateway, \Helper\Functional]

Пересоздаем базовые классы тестов:

codecept build

Теперь можем приступать к написанию тестов.

С помощью генератора добавим тест для страницы авторизации:

codecept generate:cept functional AuthPage

В созданном файле (tests/functional/AuthPageCept.php) пишем следующий код:

$I = new FunctionalTester($scenario);

$I->amOnPage('/');

$I->click('Авторизация');

$I->fillField('username','Admin');

$I->fillField('password','1q2w3e');

$I->click('Войти');

$I->see('Добро пожаловать, Admin', 'h1');

Запускаем функциональные тесты командой:

codecept run functional

При успешном выполнении увидим такой результат:

Functional Tests (1) ------------------------------

✔ AuthPageCept: (0.03s)

---------------------------------------------------

Time: 165 ms, Memory: 14.25MB

OK (1 test, 1 assertion)

## Functional Testing,(функциональное тестирование)

Тестирование кода PHP, взаимодействующего с БД

Процесс написания тестов для кода, взаимодействующего с БД, практически не отличается от процедуры тестирования обычных классов PHP. Сначала необходимо создать тестовый класс, наследующий PHPUnit\_Extensions\_Database\_TestCase (класс PHPUnit\_Extensions\_Database\_TestCase сам при этом наследует PHPUnit\_Framework\_TestCase), который будет содержать тесты для методов тестируемого класса. Затем создать тестовые методы, начинающиеся с префикса test, а потом скормить этот код phpunit с указанием имени тестового класса. Отличия заключаются лишь в том, что в тестовом классе обязательно необходимо реализовать два публичных метода — getConnection() и getDataSet(). Первый метод необходим для того, чтобы научить DbUnit работать с БД (придется использовать [PDO](http://php.net/manual/en/book.pdo.php)), а второй для того, чтобы сообщить фреймворку, в какое состояние переводить базу данных перед выполнением очередного теста. Под DataSet в терминологии DbUnit понимается набор из одной или более таблиц.

Как говорилось выше, перед выполнением очередного теста (представленного методом в тестовом классе), PHPUnit вызывает специальный метод setUp(), чтобы эмулировать среду выполнения для объекта тестируемого класса. В случае DbUnit реализация по умолчанию метода setUp() уже не пустая. Если говорить в общих чертах, то внутри метода setUp() будет создан некий объект databaseTester, который, используя определенный нами метод getConnection(), переведет базу в состояние, представленное набором таблиц (DataSet`ом), получаемым при вызове метода getDataSet(). Если вы были внимательны, то реализация метода getDataSet() также должна предоставляться тестовым классом, т.е. нами. В результате получим похожую последовательность вызовов

setUp() {/\* Установили БД в соответствии с данными, получаемыми от метода getDataSet() \*/}

testMethod1() {/\* протестировали метод 1 класса \*/}

tearDown() {/\* Очистили систему \*/}

setUp() {/\* Установили БД в соответствии с данными, получаемыми от метода getDataSet() \*/}

testMethod2() {/\* протестировали метод 2 класса \*/}

tearDown() {/\* Очистили систему \*/}

…

setUp() {/\* Установили БД в соответствии с данными, получаемыми от метода getDataSet() \*/}

testMethodN() {/\* протестировали метод N класса \*/}

tearDown() {/\* Очистили систему \*/}

Маленькие неприятности

Оперативная обстановка: База данных, используемая в проекте, имеет несколько десятков таблиц, движок MySQL InnoDB. Механизм внешних ключей активно используется с целью поддержания согласованности данных на уровне самой БД.

1. Инициализация базы

Первая неприятность, которая начала омрачать мне процесс тестирования — инициализация базы данных созданными мной наборами таблиц.

DbUnit позволяет создавать DataSet`ы, получая данные из различных источников:

Flat Xml — такой простенький способ описание состояния БД в xml-файле, рассчитанный преимущественно на ручное формирование файла.

Xml — полноценный формат задания состояния, намного больше букаф, но и более широкие возможности (можно задавать null-значения, более точно описывать структуру БД и пр.).

MySQL Xml — разновидность предыдущего формата, любезно предоставленная разработчиками DbUnit, позволяющая создавать объект DataSet на основании экспорта данных БД утилитой mysqldump.

Создание объекта DataSet по текущему состоянию БД.

Каждый из вышеперечисленных способов создания наборов таблиц реализуется отдельным методом класса PHPUnit\_Extensions\_Database\_TestCase.

Я избрал себе в помощники mysqldump и ринулся в атаку: сформировал нужное состояние базы, выгрузил его в xml и в реализации getDataSet() написал что-то вроде:

public function getDataSet() { return $this->createMySQLXMLDataSet('db\_init.xml'); //имя файла, полученного mysqldump. }

… и решил прогнать первый тест. Однако, тут же получил исключение, в котором недвусмысленно говорилось о том, что база данных не может быть приведена в заданное состояние из-за наличия в ней ограничений по внешним ключам.

Несколько минут копания в исходниках DbUnit показали, что в методе PHPUnit\_Extensions\_Database\_TestCase::setUp() установка базы в состояние в соответствии с указанным мной DataSet`ом, осуществляется при помощи операции PHPUnit\_Extensions\_Database\_Operation\_Factory::CLEAN\_INSERT. Операция CLEAN\_INSERT в свою очередь представляет собой порождаемую фабрикой макрокоманду, включающую в себя две операции: PHPUnit\_Extensions\_Database\_Operation\_Factory::TRUNCATE и PHPUnit\_Extensions\_Database\_Operation\_Factory::INSERT. Очевидно, что тут все стало на свои места — не возможно сделать TRUNCATE для базы, у которой имеются активные ограничения по внешним ключам FOREIGN KEY.

Нужно решать. Пути два — либо временно отключить FOREIGN KEY во время тестирования (темный путь), либо использовать новую команду PHPUnit\_Extensions\_Database\_Operation\_Factory::DELETE\_ALL, обнаруженную во время курения исходников DbUnit (светлый, но более длинный путь). Через минуту темная сторона во мне пересилила, и я решил пойти более простым путем — отключить ограничения целостности по внешним ключам во время создания подключения. Благо код создания все равно был написан мной в реализации метода getConnection().

Типовая реализация getConnection() выглядит примерно так:

public function getConnection() {

if (is\_null($this->m\_oConn)) {

$oPdo = new PDO('mysql:dbname=db1;host=localhost', 'root', 'qwerty');

$this->m\_oConn = $this->createDefaultDBConnection($oPdo, 'db1'); }

return $this->m\_oConn; }

$m\_oConn — это переменная-член тестового класса, которая представляет собой некоторую обертку вокруг PDO. А если быть точным, то это экземпляр класса PHPUnit\_Extensions\_Database\_DB\_DefaultDatabaseConnection. Добавив сразу после создания объекта PDO строку $oPdo->exec('SET foreign\_key\_checks = 0') я на какое-то время решил проблему с инициализацией.

Собственно, как и следовало ожидать, через некоторое время я напоролся на грабли с несогласованностью данных в базе и пришлось возвращаться на светлый путь, а именно — отказаться от отключения внешних ключей и заменить TRUNCATE на DELETE\_ALL.

Очередной просмотр исходников показал, что копать нужно в сторону реализации PHPUnit\_Extensions\_Database\_TestCase::setUp(). Вот ее код:

protected function setUp() {

parent::setUp(); //вызов PHPUnit\_Framework\_TestCase::setUp() - пустая реализация

$this->databaseTester = NULL;

$this->getDatabaseTester()->setSetUpOperation($this->getSetUpOperation());

$this->getDatabaseTester()->setDataSet($this->getDataSet());

$this->getDatabaseTester()->onSetUp(); }

и вот метод getSetUpOperation():

protected function getSetUpOperation() {

return PHPUnit\_Extensions\_Database\_Operation\_Factory::CLEAN\_INSERT(); }

Переопределив в своем тестовом классе метод getSetUpOperation() на:

protected function getSetUpOperation() {

return PHPUnit\_Extensions\_Database\_Operation\_Factory::INSERT(); }

я избавился от TRUNCATE, но добавил себе необходимость реализации очистки базы данных. Так как наша база содержит несколько представлений, то бездумный вызов PHPUnit\_Extensions\_Database\_Operation\_Factory::DELETE\_ALL() для DataSet`а из всех таблиц базы ни к чему хорошему не привел бы. К тому же я посчитал, что функциональность очистки базы может быть достаточно полезной не только в момент инициализации теста, поэтому решил оформить ее в виде самостоятельного метода:

protected function clearDb() {

$aTableNames = $this->getConnection()->createDataSet()->getTableNames();

foreach ($aTableNames as $i => $sTableName) {

if (false === strpos($sTableName, 'view\_'))

continue;

unset($aTableNames[$i]);

}

$aTableNames = array\_values($aTableNames);

$op = PHPUnit\_Extensions\_Database\_Operation\_Factory::DELETE\_ALL();

$op->execute($this->getConnection(), $this->getConnection()->createDataSet($aTableNames)); }

В коде делается допущение, что все представления, существующие в базе начинаются с префикса view\_.   
Осталось только переопределить метода setUp(), чтобы он самостоятельно очищал базу перед тем, как отдавать ее на заполнение данными databaseTester`у.

protected function setUp() {

$this->clearDb(); parent::setUp(); }

2. Сравнение наборов таблиц

Следующая проблема возникла при попытке сравнения двух DataSet`ов — одного полученного непосредственно из базы (сформированного в результате выполнения тестируемого кода), а другого — созданного заранее руками и представляющего желаемый результат.

Текущее состояние базы можно получить следующим способом:

$oActualDataSet = $this->getConnection()->createDataSet();

Увидев в манах метод PHPUnit\_Extensions\_Database\_TestCase::assertDataSetsEqual, сравнивающий два набора таблиц я очень обрадовался. Как оказалось рановато. Результаты сравнения оказались весьма неожиданными. Два идентичных на вид набора таблиц при сравнении вызывали падение assert`а.

Отладчик в свою очередь показал, что беда в DataSet`е, получаемом из базы. Видимо в целях оптимизации, при вызове $this->getConnection()->createDataSet() в тестовом классе, происходит лишь частичная загрузка набора таблиц, а если быть точным — только метаданные DataSet`а (имя базы и еще какая-то шелуха).

Исходный код PHPUnit\_Extensions\_Database\_TestCase::assertDataSetsEqual следующий:

public static function assertDataSetsEqual(PHPUnit\_Extensions\_Database\_DataSet\_IDataSet $expected, PHPUnit\_Extensions\_Database\_DataSet\_IDataSet $actual, $message = '') {

$constraint = new PHPUnit\_Extensions\_Database\_Constraint\_DataSetIsEqual($expected);

self::assertThat($actual, $constraint, $message); }

Если раскручивать цепочку вызовов дальше, то после нескольких делегирований непосредственно операции сравнения дело дойдет до PHPUnit\_Extensions\_Database\_DataSet\_AbstractTable::matches(PHPUnit\_Extensions\_Database\_DataSet\_ITable $other), в котором будут сравниваться две таблицы. В этом методе при сравнении таблиц данные в них будут в обязательном порядке затянуты из базы. Но это если дело дойдет до этого метода. Потому что прежде чем сравнивать таблицы двух DataSet`ов между собой, производится сравнения DataSet`ов. В итоге assert в каком-то месте не проходит. Этот баг есть в issues PHPUnit/DbUnit на github, ему уже несколько месяцев.

В ожидании исправления этой ошибки я быстренько накидал метод сравнения наборов таблиц. Не совсем в духе DbUnit, где все сделано универсальной последовательностью вызовов evaluate -> matches конкретных реализаций сравниваемых объектов, но зато рабочий:

public function compareDataSets(PHPUnit\_Extensions\_Database\_DataSet\_IDataSet $expected,

Здесь, как мне кажется, есть смысл и перейти ко вполне конкретной теории. А именно, описать какими же assert-методами мы располагаем для проверки поведения тестируемых сценариев.  
  
Два самых простых — это **assertFalse()** и **assertTrue()**. Проверяют, является ли полученное значение false и true соответственно. Далее идут уже упомянутый **assertEquals()** и обратный ему **assertNotEquals()**. В их использовании есть нюансы. Так при сравнении чисел с плавающей точкой есть возможность указать точность сравнения. Так же эти методы используются для сравнения экземпляров класса DOMDocument, массивов и любых объектов (в последнем случае равенство будет установлено, если атрибуты объектов содержат одинаковые значения). Так же следует упомянуть **assertNull()** и **assertNotNull()** которые проверяют соответствие параметра типу данных NULL (да-да, не забываем, что в PHP это отдельный тип данных). Этим возможные сравнения не ограничиваются. Нет смысла в рамках этой статьи заниматься перепечаткой документации, потому приведу по возможности структурированный список всех возможных методов. Более детально интересующиеся могут прочитать [здесь](http://www.phpunit.de/manual/current/en/api.html#api.assert)  
  
Базовые методы сравнения  
assertTrue() / assertFalse()  
assertEquals() / assertNotEquals()  
assertGreaterThan()  
assertGreaterThanOrEqual()  
assertLessThan()  
assertLessThanOrEqual()  
assertNull() / assertNotNull()  
assertType() / assertNotType()  
assertSame() / assertNotSame()  
assertRegExp() / assertNotRegExp()  
  
Методы сравнения массивов  
assertArrayHasKey() / assertArrayNotHasKey()  
assertContains() / assertNotContains()  
assertContainsOnly() / assertNotContainsOnly()  
  
ООП специфичные методы  
assertClassHasAttribute() / assertClassNotHasAttribute()  
assertClassHasStaticAttribute() / assertClassNotHasStaticAttribute()  
assertAttributeContains() / assertAttributeNotContains()  
assertObjectHasAttribute() / assertObjectNotHasAttribute()  
assertAttributeGreaterThan()  
assertAttributeGreaterThanOrEqual()  
assertAttributeLessThan()  
assertAttributeLessThanOrEqual()  
  
Методы сравнения файлов  
assertFileEquals() / assertFileNotEquals()  
assertFileExists() / assertFileNotExists()  
assertStringEqualsFile() / assertStringNotEqualsFile()  
  
Методы сравнения XML  
assertEqualXMLStructure()  
assertXmlFileEqualsXmlFile() / assertXmlFileNotEqualsXmlFile()  
assertXmlStringEqualsXmlFile() / assertXmlStringNotEqualsXmlFile()  
assertXmlStringEqualsXmlString() / assertXmlStringNotEqualsXmlString()  
  
Разное  
assertTag()  
assertThat()

### Генераторы

Codeception имеет множество полезных команд для генерации всего и вся.

* generate:cept suite filename - Генерировать Cept сценарий.
* generate:cest suite filename - Генерировать Cest тест.
* generate:test suite filename - Генерировать PHPUnit Test c Codeception хуками.
* generate:phpunit suite filename - Генерировать классический PHPUnit Test.
* generate:suite suite guy - Генерировать новый набор тестов с указанным Guy-классом.
* generate:scenarios suite - Генерировать текстовый файл содержащий сценарии из тестов.

# Функциональное тестирование

Теперь, когда мы имеем написанные приемочные тесты, настало время рассмотреть функциональные тесты. Функциональные тесты это почти то же самое, что и приемочные, однако есть одно существенное различие: они не требуют использования веб сервера для запуска своих сценариев. Другими словами мы будем запускать наше приложение внутри тестов, имитируя запросы и ответы.

Говоря простыми словами, мы устанавливаем переменные $\_REQUEST, $\_GET и $\_POST, затем выполняем скрипт внутри теста, получаем ответ и тестируем все это.  
Функциональное тестирование часто может быть более лучшим решением, чем приемочное, потому как такие тесты не требуют использования веб сервера и могут предложить более подробный отладочный вывод. К примеру, если ваш сайт выбросит исключение, оно будет напечатано в консоли.

Codeception может подключаться к различным веб фреймворкам, поддерживающим функциональное тестирование. К примеру, вы можете запустить функциональные тесты для приложения построенного поверх Zend Framework, Symfony или Symfony2, используя лишь модули поставляемые Codeception! Список поддерживаемых модулей будет разобран позже.

Модули для всех этих фреймворков имеют одинаковый интерфейс, поэтому ваши тесты не будут связаны друг с другом. Вот простой пример функционального теста.

<?php

$I = new TestGuy($scenario);

$I->amOnPage('/');

$I->click('Login');

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

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

$I->click('Enter');

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

// $I->seeEmailIsSent() - special for Symfony2

?>

Такой же тест, как и приемочный. Как видите, можно использовать одинаковые методы и для приемочных, и для функциональных тестов.  
Мы рекомендуем тестировать нестабильные части приложения с помощью функциональных тестов, а стабильные с помощью приемочных.

## Ловушки

Приемочные тесты обычно намного медленнее, чем функциональные. Однако функциональные тесты менее стабильны и запускают тестовый фреймворк и приложение в одном окружении.

#### Headers, Cookies, Sessions

Одна из известных проблем функциональных тестов - использование PHP функций работающих с переменными из категории headers,sessions, cookies.  
Как вы знаете, функция header возвратит ошибку, если будет выполнена более одного раза. В функциональных тестах мы запускаем наше приложение несколько раз, таким образом мы получим много ненужных ошибок при отображении результатов.

#### Разделяемая память

При функциональном тестировании, в отличие от традиционного, приложение PHP не останавливается после выполнения запроса.  
Так как все запросы выполняются в одном контейнере памяти, они не изолированны.  
Таким образом **если вы заметили, что ваши тесты магическим образом падают, однако не должны - попробуйте выполнить один тест.**  
Это проверит, изолированы ли тесты во время работы. Потому что довольно просто поломать окружение, когда все тесты выполняются в разделяемой памяти.  
Держите память в чистоте, избегайте утечек памяти и очищайте глобальные и статические переменные.

## Основы функционального тестирования

Ваши функциональные тесты располагаются в каталоге tests/functional.  
Для начала вам необходимо включить один из модулей фреймворков в конфигурационный файл тестового набора:tests/functional.suite.yml.  
Примеры конфигурации фреймворков описаны ниже в данной главе.

После вам необходимо пересобрать Guy-классы

php codecept.phar build

Для генерации теста вы можете использовать стандартную команду генератор generate:cept:

php codecept.phar generate:cept functional myFirstFunctional

После чего выполнить тесты с помощью run:

php codecept.phar run functional

Используйте опцию --debug для более детального вывода.

## Сообщения об ошибках

По умолчанию Codeception использует значение E\_ALL & ~E\_STRICT & ~E\_DEPRECATED.  
В функциональных тестах вы можете захотеть сменить эти значения в зависимости от используемого фреймворка.  
Сообщения об ошибках могут быть настроены в конфигурационном файле набора:

class\_name: TestGuy

modules:

enabled: [Yii1, TestHelper]

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

error\_level может быть установлен глобально в файле codeception.yml.

## Фреймворки

Codeception интегрирован с большинством популярных PHP фреймворков.  
Мы нацелены на то, чтобы иметь модули для большинства популярных библиотек.  
Помогите нам в разработке, если вы используете фреймворк, которого нет в списке.

### Symfony2

Для интеграции Symfony2 вам не нужно устанавливать никаких бандлов или менять конфигурацию.  
Просто подключите модуль Symfony2 в ваш тестовый набор. Если вы используете Doctrine2, не забудьте подключить его тоже.

Пример для functional.suite.yml

class\_name: TestGuy

modules:

enabled: [Symfony2, Doctrine2, TestHelper]

По умолчанию модуль будет искать ядро в директории app.

Модуль использует Symfony Profiler для предоставления дополнительной информации.

[Смотрите полную справку](http://codeception.com/docs/modules/Symfony2)

### Laravel 4

[Laravel](http://codeception.com/docs/modules/Laravel4) не имеет конфигурации и так же может быть легко настроен.

class\_name: TestGuy

modules:

enabled: [Laravel4, TestHelper]

### Yii

Сам по себе Yii framework не имеет движка для функционального тестирования.  
Таким образом, Codeception первый и единственный фреймворк для функционального тестирования на Yii.  
Для использования его с Yii включите модуль Yii1 в файл конфигурации.

class\_name: TestGuy

modules:

enabled: [Yii1, TestHelper]

Для того, чтобы избежать подводных камней, описаных ранее, Codeception предоставляет некоторые хуки поверх движка Yii.  
Для его установки [следуйте шагам описаным в руководстве](http://codeception.com/docs/modules/Yii1).

### Zend Framework 2

Используйте модуль [ZF2](http://codeception.com/docs/modules/ZF2) для запуска функциональных тестов внутри Zend Framework 2.

class\_name: TestGuy

modules:

enabled: [ZF2, TestHelper]

### Zend Framework 1.x

Модуль для Zend Framework значительно заимствует функционал класса ControllerTestCase, используемого для функциональных тестов с помощью PHPUnit.  
Он использует похожие подходы для загрузки и очистки. Для использования Zend Framework в ваших функциональных тестах включите модуль ZF1.

Пример для functional.suite.yml

class\_name: TestGuy

modules:

enabled: [ZF1, TestHelper]

[Смотрите полную справку](http://codeception.com/docs/modules/ZF1)

### symfony

Этот модуль один из первых, который был разработан для Codeception. Поэтому его действия могут отличаться от действий, используемых в других фреймворках.  
Он предоставляет множество удобных операций, таких как логирование с помощью sfGuardAuth или валидация внутри теста.

Пример для functional.suite.yml

class\_name: TestGuy

modules:

enabled: [Symfony1, TestHelper]

[Смотрите полную справку](http://codeception.com/docs/modules/Symfony1)