Testing application with Codeception

By default, the basic and advanced Yii2 application skeletons use Codeception as a testing framework.  
Codeception supports writing of unit, functional, and acceptance tests out of the box. For unit tests, it uses  
the PHPUnit test framework, which will be covered in the next recipe.

Getting ready

1. Create a new yii2-app-basic application using the Composer package manager, as described in  
   the official guide at [http://www.yiiframework.com/doc-2.0/guide-start-insta]lation.html](http://www.yiiframework.com/doc-2.0/guide-start-installation.html).

Note

Note: If your use version 2.0.9 (or earlier) of the basic application just upgrade manually tests  
directory, and also add config/test .php, config/test\_db .php and web/index-test.php files.  
Besides you must copy require and require-dev sections of composer .json file and run  
composer updat e.

1. Create and apply the following migration:

<?php

use yii\db\Migration;

class m160309\_070856\_create\_post extends Migration  
{

public function up()

{

$this->createTable('{{%post}}', [

'id' => $this->primaryKey(),

'title' => $this->string()->notNull(),

'text' => $this->text()->notNull(),

'status' => $this->smallInteger()->notNull()-  
>defaultValue(0),

]);

}

public function down()

{

$this->dropTable('{{%post}}');

}

}

1. Create the Post model:  
   namespace app\models;  
   use Yii;

use yii\db\ActiveRecord;

/\*\*

* @property integer $id
* @property string $title
* @property string $text
* @property integer $status
* @property integer $created\_at
* @property integer $updated\_at  
  \*/

class Post extends ActiveRecord  
{

const STATUS\_DRAFT = 0;  
const STATUS\_ACTIVE = 1;

public static function tableName()

{

return '{{%post}}';

}

public function rules()

{

return [

[['title', 'text'], 'required'],

[ [' text'], 'string'],

['status', 'in', 'range' => [self::STATUS\_DRAFT, self::STATUS\_ACTIVE]],  
['status', 'default', 'value' => self::STATUS\_DRAFT],

[['title'], 'string', 'max' => 255],

];

}

public function behaviors()

{

return [

TimestampBehavior::className(),

];

}

public static function getStatusList()

{

return [

self::STATUS\_DRAFT => 'Draft',  
self::STATUS\_ACTIVE => 'Active',

];

}

public function publish()

{

if ($this->status == self::STATUS\_ACTIVE) {

throw new \DomainException('Post is already published.');

}

$this->status = self::STATUS\_ACTIVE;

}

public function draft()

{

if ($this->status == self::STATUS\_DRAFT) {

throw new \DomainException('Post is already drafted.');

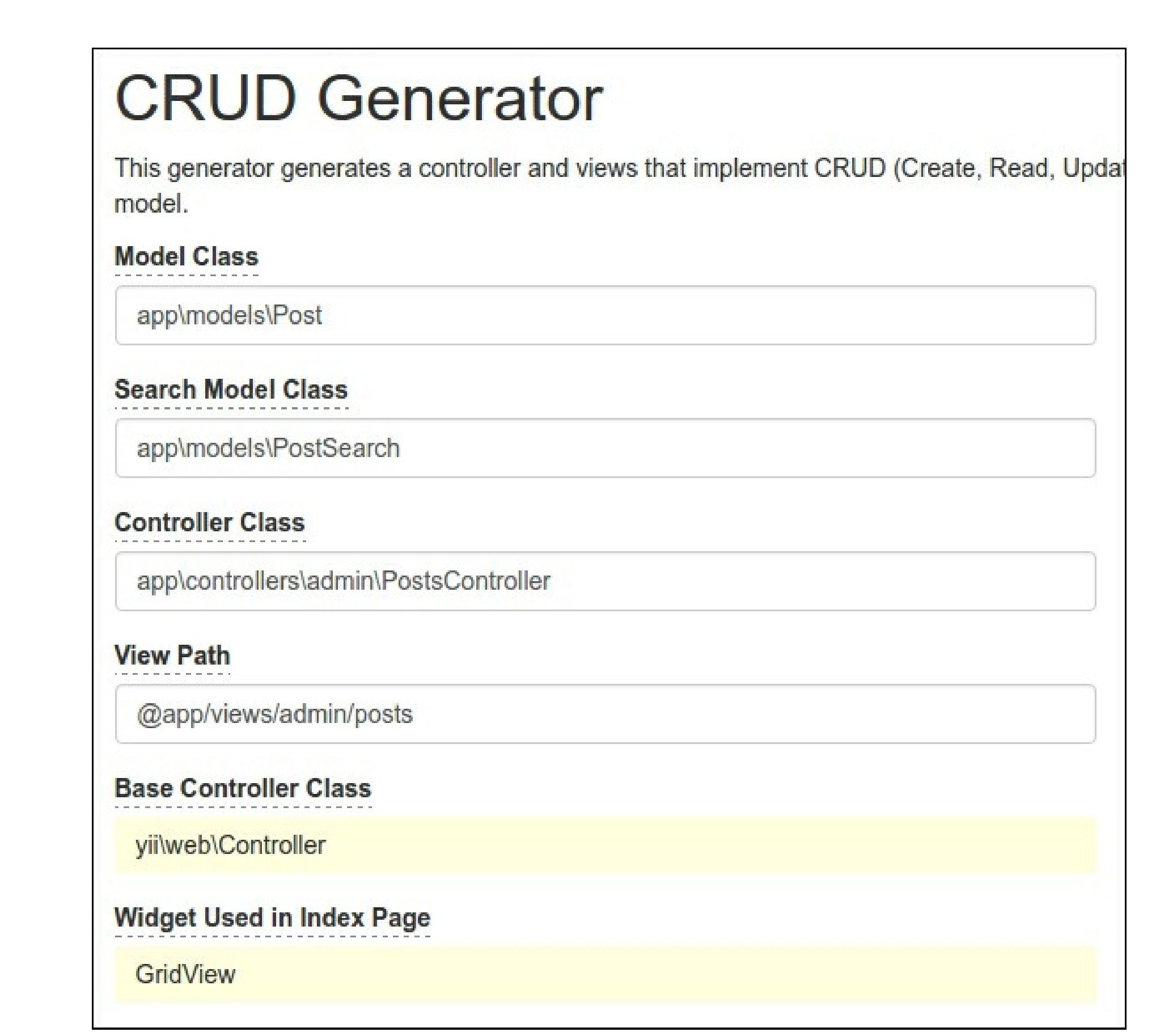
}

$this->status = self::STATUS\_DRAFT;

}

}

4. Generate CRUD:



5. Also, add the status drop-down list for the status field and name for the submit button in  
views/admin/posts/\_form.php:

<div class="post-form">

<?php $form = ActiveForm::begin(); ?>

<?= $form->field($model, 'title')->textInput(['maxlength' => true]) ?>

<?= $form->field($model, 'text')->textarea(['rows' => 6]) ?>

<?= $form->field($model, 'status')->dropDownList(Post::getStatusList()) ?>  
<div class="form-group">

<?= Html::submitButton($model->isNewRecord ? 'Create' : 'Update', [

'class' => $model->isNewRecord ? 'btn btn-success' : 'btn btn-primary',  
'name' => 'submit-button',

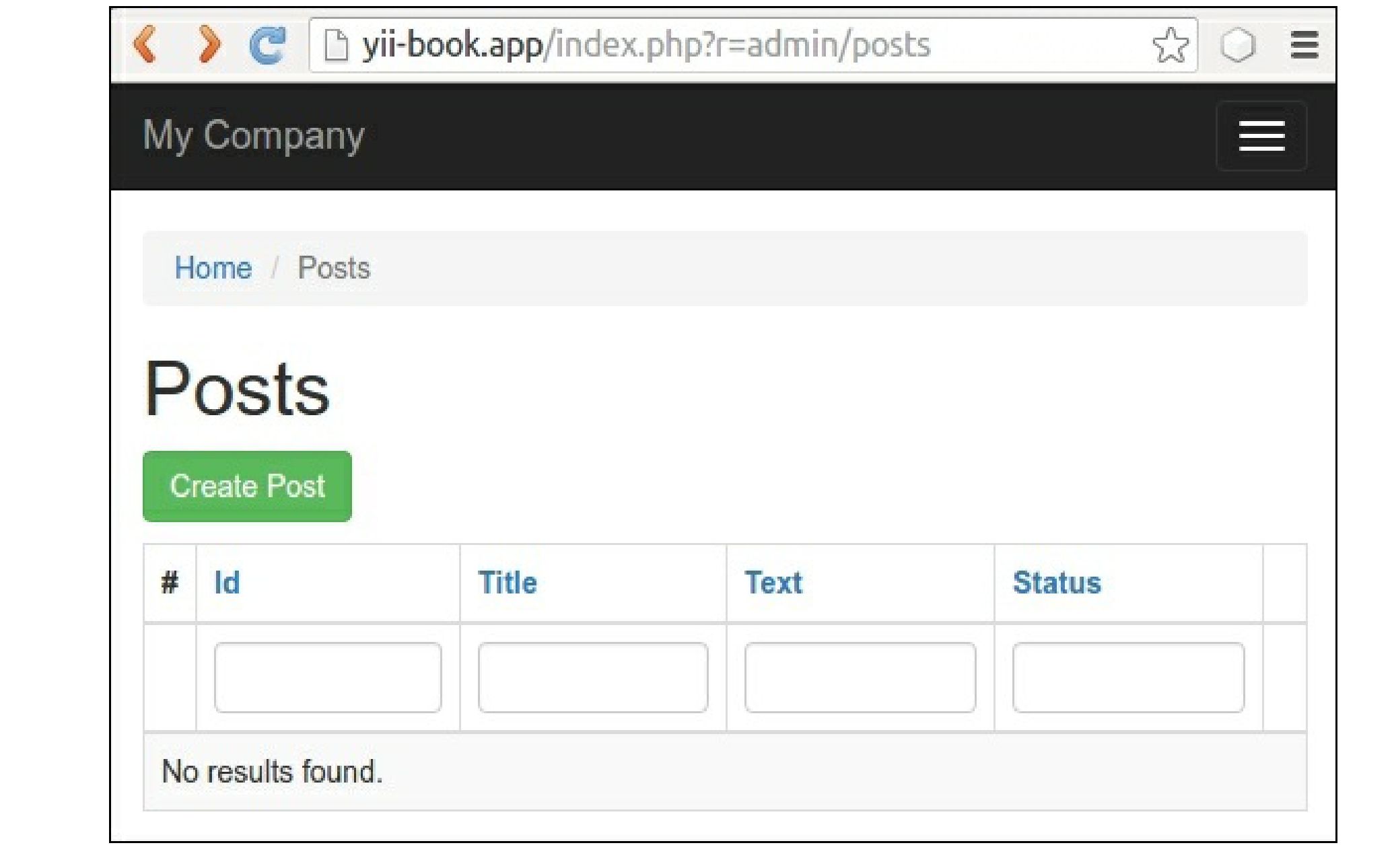
]) ?>

</div>

<?php ActiveForm::end(); ?>

</div>

6. Now check that the controller works:



Create any demo posts.

How to do it...

Preparing for the tests

Follow these steps to prepare for the tests:

1. Create yii2\_basic\_tests or other test database and update it by applying migrations:  
tests/bin/yii migrate

The command needs to be run in the tests directory. You can specify your test database options in  
configuration file /config/test\_db. php.

2. Codeception uses autogenerated Actor classes for own test suites. Build them with this command:

composer exec codecep  
t build

Running unit and functional tests

We can run any types of the application’s tests right now:

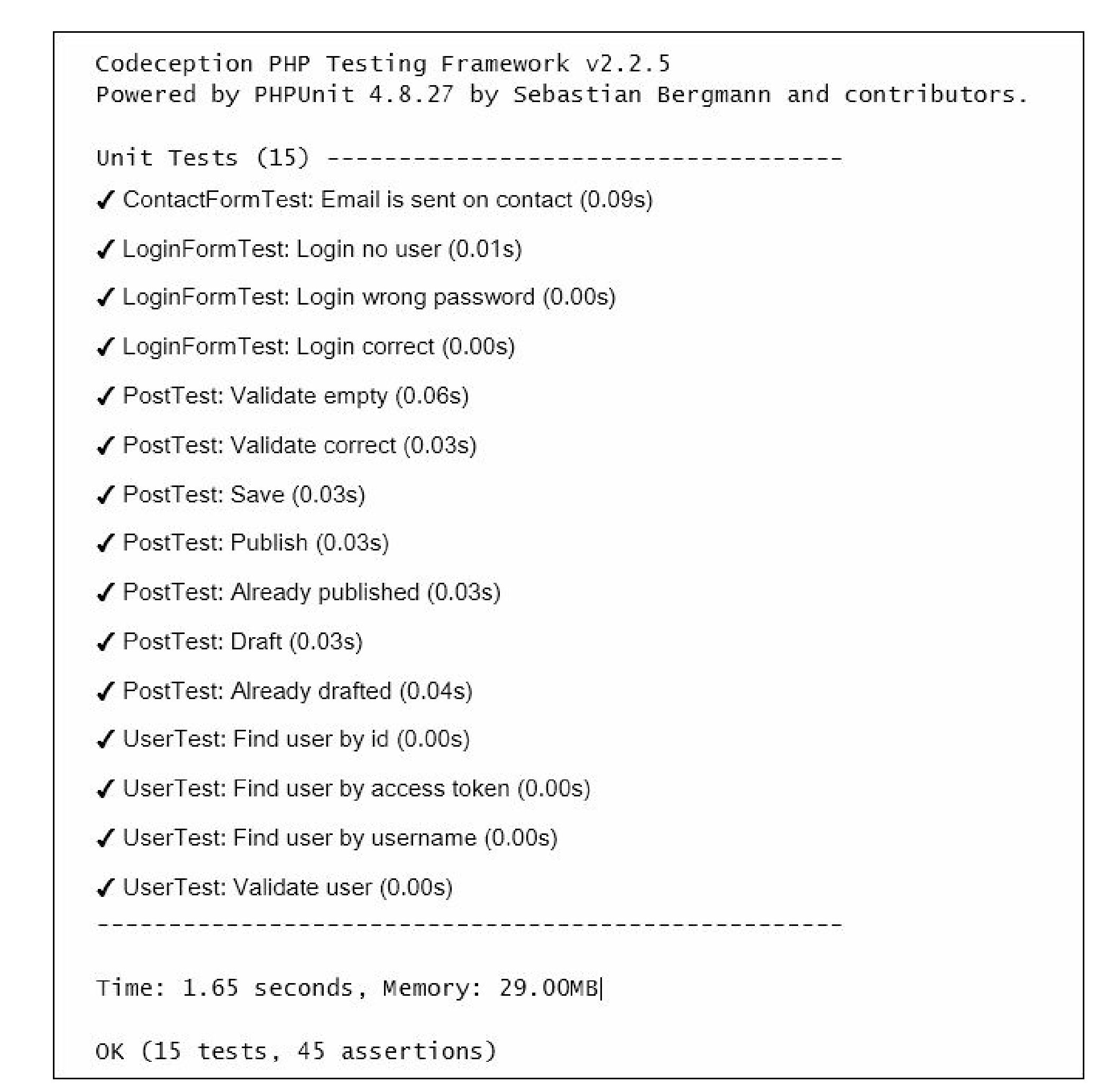
* run all available tests  
  composer exec codecept run
* run functional tests

composer exec codecept run functional

* run unit tests

composer exec codecept run unit

As a result, you can view, testing report like this:



Getting coverage reports

You can get code coverage reports for your code. By default, code coverage is disabled in the  
tests/codeception. yml configuration file; you should uncomment the necessary rows to be able to  
collect code coverage:

coverage:

enabled: true  
whitelist:  
include:

* models/\*
* controllers/\*
* commands/\*
* mail/\*  
  blacklist:

include:

* assets/\*
* config/\*
* runtime/\*
* vendor/\*
* views/\*
* web/\*
* tests/\*

You must install the XDebug PHP extension from <https://xdebug.org>. For example, on Ubuntu or Debian  
you can type the following in your terminal:

sudo apt-get install php5-xdebug

On Windows, you must open the php. ini file and add the custom code with the path to your PHP  
installation directory:

[xdebug]

zend\_extension\_ts=C:/php/ext/php\_xdebug.dll

Alternatively, if you use the non-thread safe edition, type the following:

[xdebug]

zend\_extension=C:/php/ext/php\_xdebug.dll

Finally, you can run tests and collect the coverage report with the following command:

#collect coverage for all tests

composer exec codecept run --coverage-html

#collect coverage only for unit tests  
composer exec codecept run unit --coverage-html

#collect coverage for unit and functional tests

composer exec codecept run functional,unit --coverage-html

You can see the text code coverage output in the terminal:

Code Coverage Report:

2016-03-31 08:13:05

Summary:

Classes: 20.00% (1/5)

Methods: 40.91% (9/22)

Lines: 30.65% (38/124)

\app\models::ContactForm

Methods: 33.33% ( 1/ 3) Lines: 80.00% ( 12/ 15)

\app\models::LoginForm

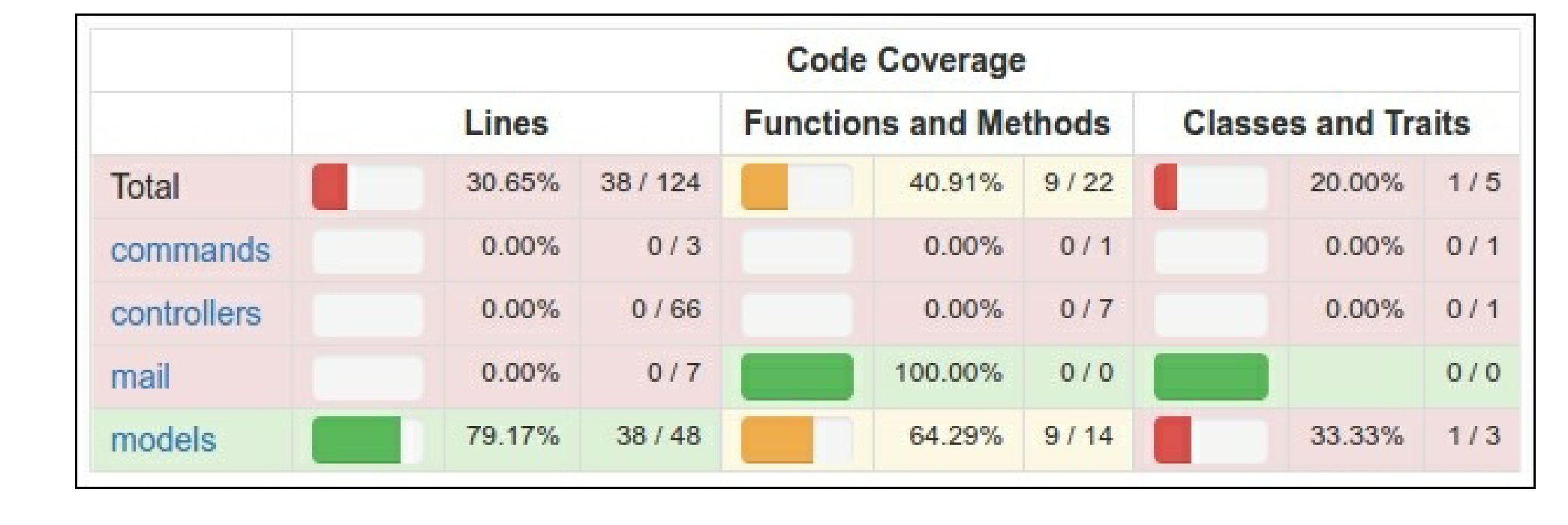
Methods: 100.00% ( 4/ 4) Lines: 100.00% ( 18/ 18)

\app\models::User

Methods: 57.14% ( 4/ 7) Lines: 53.33% ( 8/ 15)

Remote CodeCoverage reports are not printed to console  
HTML report generated in coverage

Also, you can see HTML-report under the tests/codeception/\_output/coverage directory:



You can click on any class and analyze which lines of code have not been executed during the testing  
process.

Running acceptance tests

In acceptance tests you can use PhpBrowser for requesting server via Curl. It helps to check your site  
controllers and to parse HTTP and HTML response codes. But if you want to test your CSS or JavaScript  
behavior, you must use real browser.

Selenium Server is an interactive tool, which integrates into Firefox and other browsers and allows to open  
site pages and emulate human actions.

For working with real browser we must install Selenium Server:

1. Require full Codeception package instead of basic:

composer require --dev codeception/codeception  
composer remove --dev codeception/base

1. Download the following software:

° Install Mozilla Firefox browser from <https://www.mozilla.org>  
o Install Java Runtime Environment from <https://www.java.com/en/download/>

° Download Selenium Standalone Server from <http://www.seleniumhq.org/download/>

° Download Geckodriver from <https://github.com/mozilla/geckodriver/releases>

1. Launch server with the driver in new terminal window:

java -jar -Dwebdriver.gecko.driver=~/geckodriver ~/selenium-server-standalone-  
x.xx.x.jar

1. Copy tests/acceptance . suite . yml. example to tests/acceptance . suite . yml file and configure  
   one like this:

class\_name: AcceptanceTester  
modules:  
enabled:

* WebDriver:

url: <http://127.0.0.1:8080/>  
browser: firefox

* Yii2:

part: orm

entryScript: index-test.php  
cleanup: false

1. Open new terminal frame and start web server:

tests/bin/yii serve

1. Run acceptance tests:

composer exec codecept run acceptance

And you should see how Selenium starts the browser and check all site pages.

Creating database fixtures

Before running own tests, we must clear the own test database and load specific test data into it. The  
yii2-codeception extension provides the ActiveFixture base class for creating test data sets for own  
models. Follow these steps to create database fixtures:

1. Create the fixture class for the Post model:

<?php

namespace tests\fixtures;

use yii\test\ActiveFixture;

class PostFixture extends ActiveFixture  
{

public $modelClass = 'app\modules\Post';  
public $dataFile = '@tests/\_data/post.php';

}

1. Add a demonstration data set in test/\_data/post. php file:

<?php  
return [

[

'id' => 1,

'title' => 'First Post',

'text' => 'First Post Text',

'status' => 1,

'created\_at' => 1457211600,

'updated\_at' => 1457211600,

],

[

'id' => 2,

'title' => 'Old Title For Updating',

'text' => 'Old Text For Updating',

'status' => 1,

'created\_at' => 1457211600,

'updated\_at' => 1457211600,

],

[

'id' => 3,

'title' => 'Title For Deleting',

'text' => 'Text For Deleting',

'status' => 1,

'created\_at' => 1457211600,

'updated\_at' => 1457211600,

],

];

1. Activate fixtures support for unit and acceptance tests. Just add fixtures part into  
   unit. suite . yml file:

class\_name: UnitTester  
modules:  
enabled:

* Asserts
* Yii2:

part: [orm, fixtures, email]

Also, add the fixtures part into acceptance . suite . yml:

class\_name: AcceptanceTester  
modules:  
enabled:

* WebDriver:

url: <http://127.0.0.1:8080/>  
browser: firefox

* Yii2:

part: [orm, fixtures]  
entryScript: index-test.php  
cleanup: false

1. Regenerate tester classes for applying these changes by the following command:  
   composer exec codecept build

Writing unit or integration test

Unit and integration tests check the source code of our project.

Unit tests check only the current class or their method in isolation from other classes and resources such  
as databases, files, and many more.

Integration tests check the working of your classes in integration with other classes and resources.

ActiveRecord models in Yii2 always use databases for loading table schema as we must create a real test  
database and our tests will be integrational.

1. Write tests for checking model validation, saving, and changing its status:

<?php

namespace tests\unit\models;

use app\models\Post;

use Codeception\Test\Unit;

use tests\fixtures\PostFixture;

class PostTest extends Unit  
{

/\*\*

\* @var \UnitTester  
\*/

protected $tester;

public function \_before()

{

$this->tester->haveFixtures([

'post' => [

'class' => PostFixture::className(),

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

]

]);

}

public function testValidateEmpty()

{

$model = new Post();

expect('model should not validate', $model->validate())->false();

expect('title has error', $model->errors)->hasKey('title');  
expect('title has error', $model->errors)->hasKey('text');

}

public function testValidateCorrect()

{

$model = new Post([

'title' => 'Other Post',

'text' => 'Other Post Text',

]);

expect('model should validate', $model->validate())->true();

}

public function testSave()

{

$model = new Post([

'title' => 'Test Post',

'text' => 'Test Post Text',

]);

expect('model should save', $model->save())->true();

expect('title is correct', $model->title)->equals('Test Post');  
expect('text is correct', $model->text)->equals('Test Post Text');  
expect('status is draft', $model->status)->equals(Post::STATUS\_DRAFT);  
expect('created\_at is generated', $model->created\_at)->notEmpty();  
expect('updated\_at is generated', $model->updated\_at)->notEmpty();

}

public function testPublish()

{

$model = new Post(['status' => Post::STATUS\_DRAFT]);

expect('post is drafted', $model->status)->equals(Post::STATUS\_DRAFT);  
$model->publish();

expect('post is published', $model->status)->equals(Post::STATUS\_ACTIVE);

}

public function testAlreadyPublished()

{

$model = new Post(['status' => Post::STATUS\_ACTIVE]);

$this->setExpectedException('\LogicException');

$model->publish();

}

public function testDraft()

{

$model = new Post(['status' => Post::STATUS\_ACTIVE]);

expect('post is published', $model->status)->equals(Post::STATUS\_ACTIVE);  
$model->draft();

expect('post is drafted', $model->status)->equals(Post::STATUS\_DRAFT);

}

public function testAlreadyDrafted()

{

$model = new Post(['status' => Post::STATUS\_ACTIVE]);

$this->setExpectedException('\LogicException');

$model->publish();

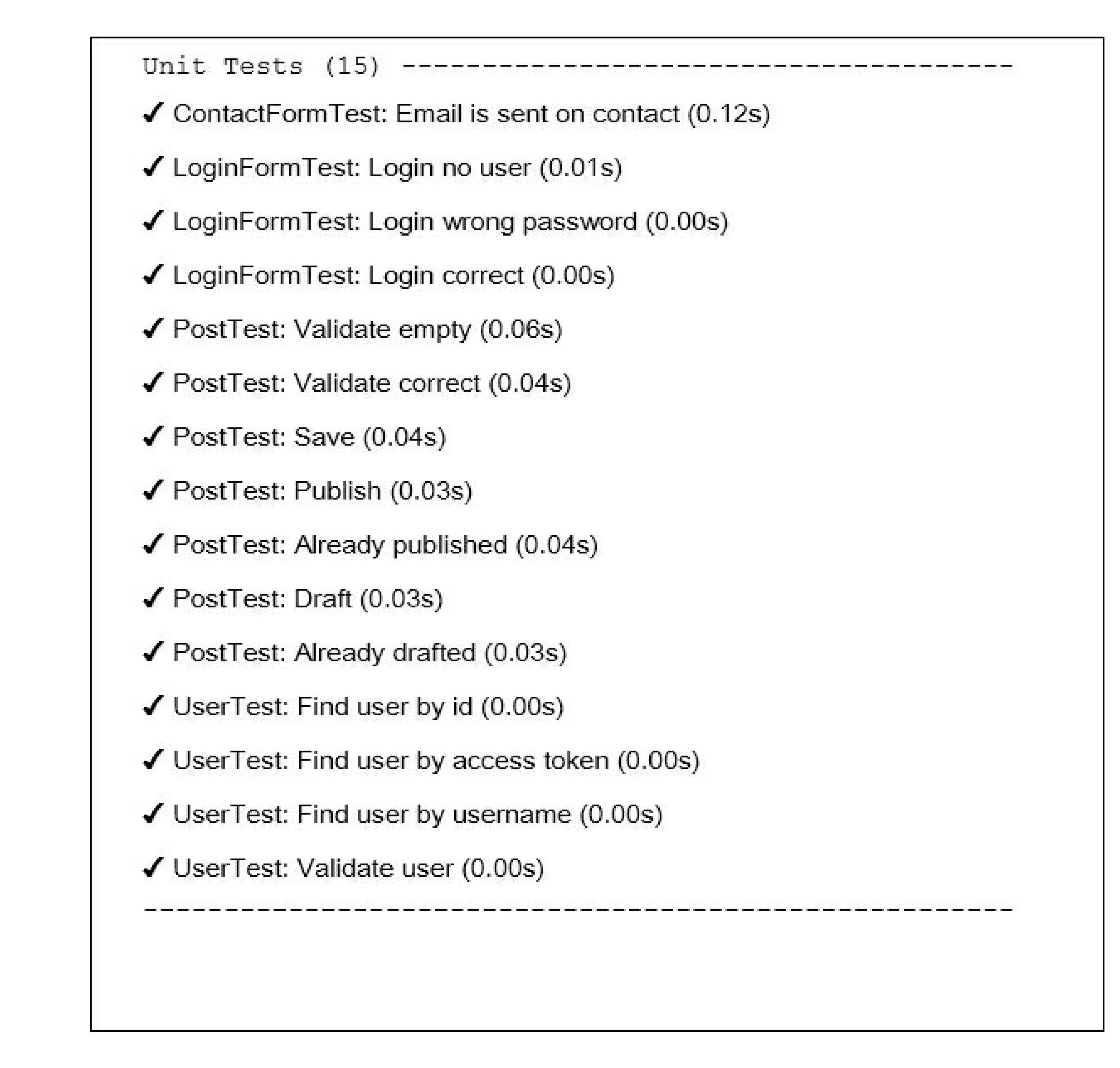
}

}

1. Run the tests:

composer exec codecept run unit

1. Now see the result:



That is all. If you deliberately or casually break any model’s method you will see a broken test.

Writing functional test

Functional test checks that your application works correctly. This suite prepares $\_get, $\_post, and others  
request variables and call the Application: : handleRequest method. It helps to test your controllers and  
their responses without running of real server.

Now we can write tests for our admin CRUD:

1. Generate a new test class:

codecept generate:cest functional admin/Posts

1. Fix the namespace in the generated file and write own tests:

<?php

namespace tests\functional\admin;

use app\models\Post;  
use FunctionalTester;  
use tests\fixtures\PostFixture;  
use yii\helpers\Url;

class PostsCest  
{

function \_before(FunctionalTester $I)

{

$I->haveFixtures([

'user' => [

'class' => PostFixture::className(),

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

]

]);

}

public function testIndex(FunctionalTester $I)

{

$I->amOnPage(['admin/posts/index']);

$I->see('Posts', 'h1');

}

public function testView(FunctionalTester $I)

{

$I->amOnPage(['admin/posts/view', 'id' => 1]);  
$I->see('First Post', 'h1');

}

public function testCreateInvalid(FunctionalTester $I)

{

$I->amOnPage(['admin/posts/create']);

$I->see('Create', 'h1');

$I->submitForm('#post-form', [

'Post[title]' => '',

' Post[text]' => '',

]);

$I->expectTo('see validation errors');

$I->see('Title cannot be blank.', '.help-block');  
$I->see('Text cannot be blank.', '.help-block');

}

public function testCreateValid(FunctionalTester $I)

{

$I->amOnPage(['admin/posts/create']);

$I->see('Create', 'h1');

$I->submitForm('#post-form', [

'Post[title]' => 'Post Create Title',

'Post[text]' => 'Post Create Text',

'Post[status]' => 'Active',

]);

$I->expectTo('see view page');

$I->see('Post Create Title', 'hi');

}

public function testUpdate(FunctionalTester $I)

{

// ...

}

public function testDelete(FunctionalTester $I)

{

$I->amOnPage(['/admin/posts/view', 'id' => 3]);

$I->see('Title For Deleting', 'h1');

$I->amGoingTo('delete item');

$I->sendAjaxPostRequest(Url::to(['/admin/posts/delete', 'id' => 3]));  
$I->expectTo('see that post is deleted');

$I->dontSeeRecord(Post::className(), [

'title' => 'Title For Deleting',

]);

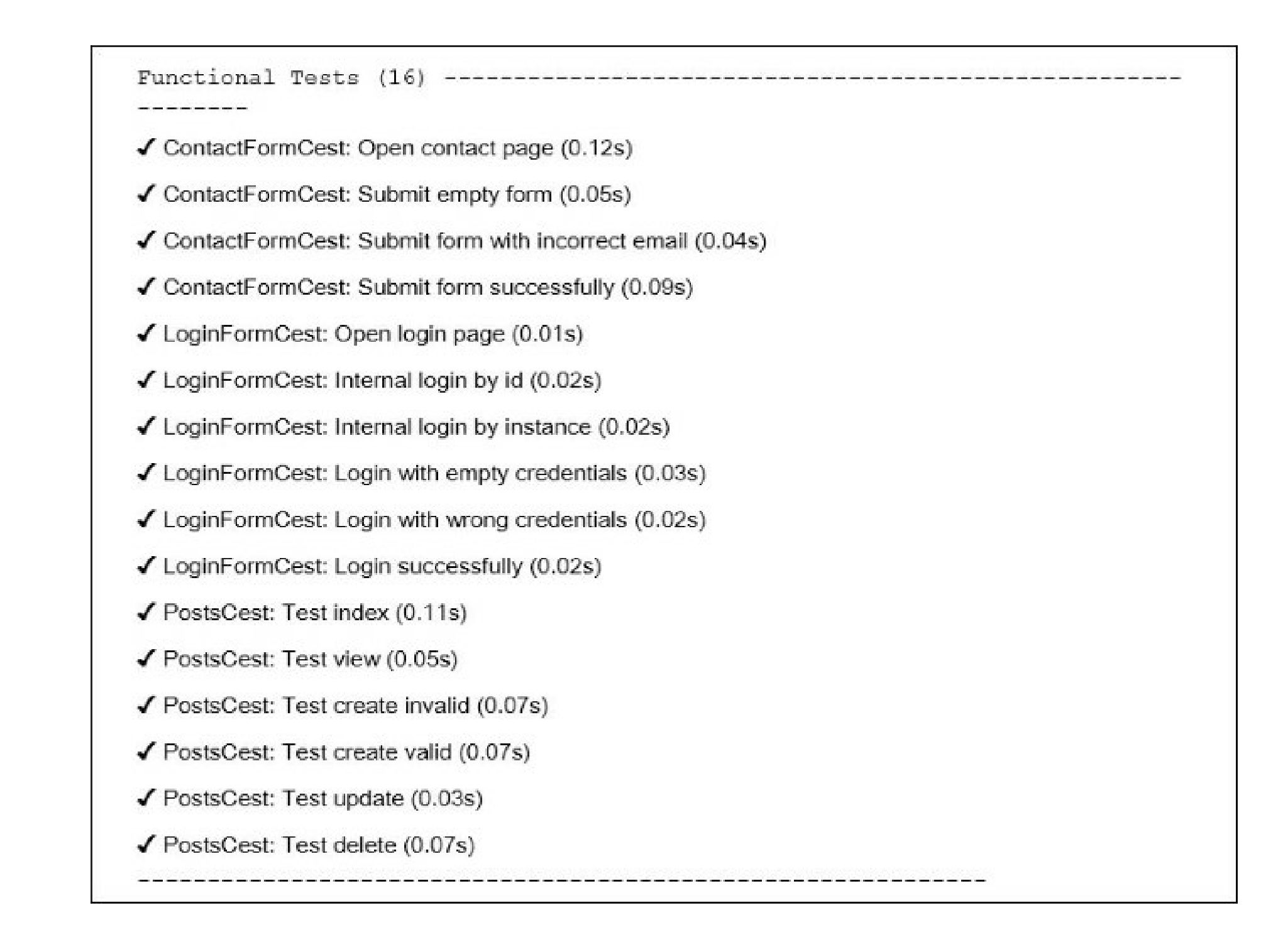
}

}

1. Run tests with the command:

composer exec codecept run functional

1. Now see the results:



All tests passed. In other case you can see snapshots of tested pages in tests/\_output directory for  
failed tests.

Writing acceptance test

1. Acceptance tester hit the real site from test server instead of calling Application: : handleRequest  
   method. High-level acceptance tests look like middle-level functional tests, but in case of Selenium  
   it allows to check JavaScript behavior in real browser.
2. You must get the following class in tests/acceptance directory:

<?php

namespace tests\acceptance\admin;

use AcceptanceTester;

use tests\fixtures\PostFixture;

use yii\helpers\Url;

class PostsCest  
{

function \_before(AcceptanceTester $I)

{

$I->haveFixtures([

'post' => [

'class' => PostFixture::className(),

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

]

]);

}

public function testIndex(AcceptanceTester $I)

{

$I->wantTo('ensure that post index page works');

$I->amOnPage(Url::to(['/admin/posts/index']));

$I->see('Posts', 'h1');

}

public function testView(AcceptanceTester $I)

{

$I->wantTo('ensure that post view page works');  
$I->amOnPage(Url::to(['/admin/posts/view', 'id' => 1]));  
$I->see('First Post', 'h1');

}

public function testCreate(AcceptanceTester $I)

{

$I->wantTo('ensure that post create page works');

$I->amOnPage(Url::to(['/admin/posts/create']));  
$I->see('Create', 'h1');

$I->fillField('#post-title', 'Post Create Title');  
$I->fillField('#post-text', 'Post Create Text');  
$I->selectOption('#post-status', 'Active');

$I->click( 'submit-button');

$I->wait(3);

$I->expectTo('see view page');

$I->see('Post Create Title', 'h1');

}

public function testDelete(AcceptanceTester $I)

{

$I->amOnPage(Url::to(['/admin/posts/view', 'id' => 3]));  
$I->see('Title For Deleting', 'h1');

$I->click('Delete');

$I->acceptPopup();

$I->wait(3);

$I->see('Posts', 'h1');

}

}

Do not forget to call wait method for waiting for page to be opened or reloaded.

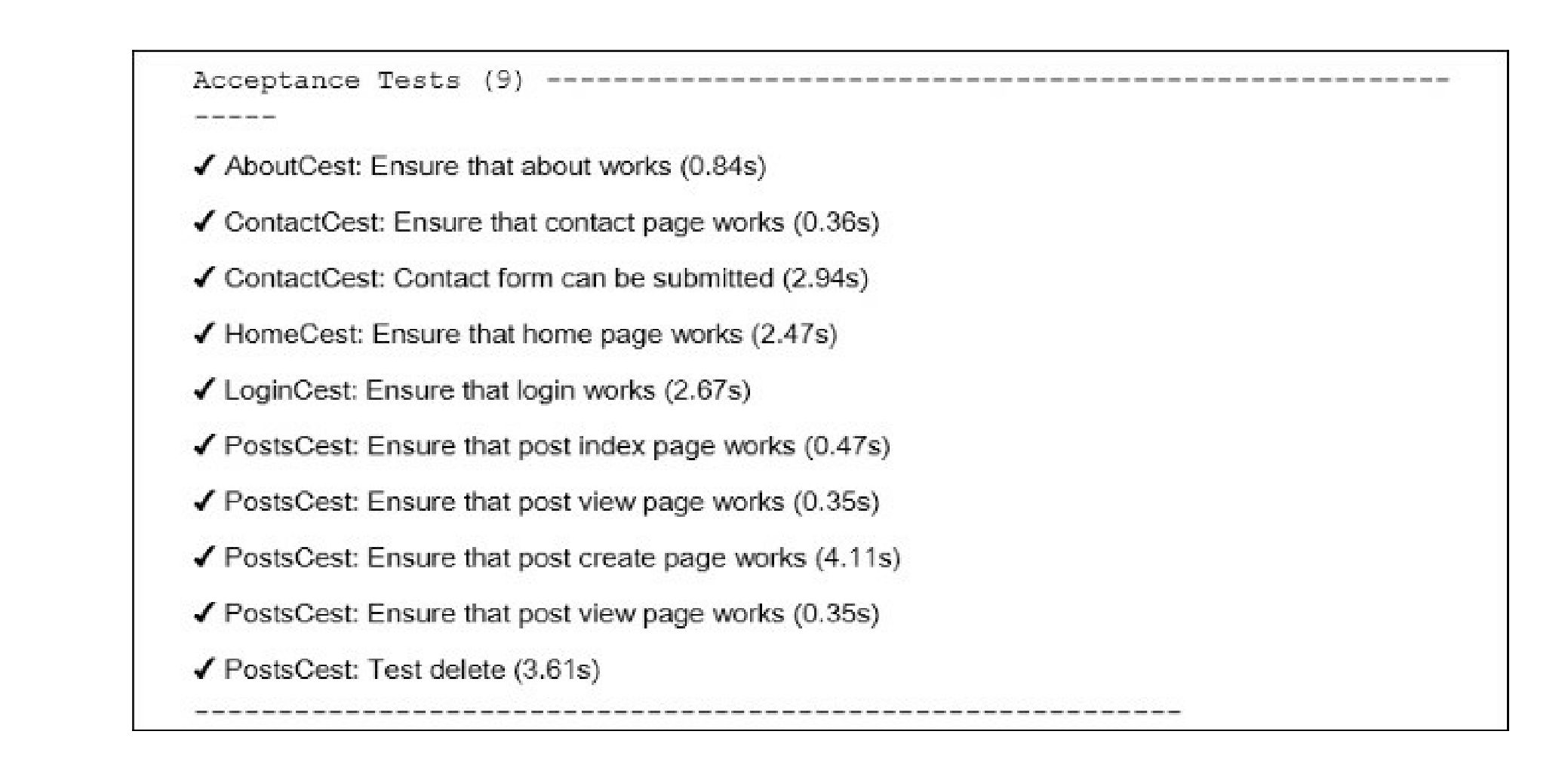
1. Run the PHP test server in a new terminal frame:

tests/bin/yii serve

1. Run the acceptance tests:

composer exec codecept run acceptance

1. See the results:



Selenium will start Firefox web browser and execute our testing commands.

Creating API test suite

Besides unit, functional, and acceptance suites, Codeception allows to create specific test suites. For  
example, we can create it for API testing with support of XML and JSON parsing.

1. Create the REST API controller controllers/api/PostsController.php for the Post model:  
   <?php

namespace app\controllers\api;

use yii\rest\ActiveController;

class PostsController extends ActiveController  
{

public $modelClass = '\app\models\Post';

}

1. Add REST routes for the UrlManager component in config/web. php:

'components' => [

// ...

'urlManager' => [

'enablePrettyUrl' => true,

'showScriptName' => false,

'rules' => [

['class' => 'yii\rest\UrlRule', 'controller' => 'api/posts'],

],

],

],

and some config (but with enabled showScriptName option) in config/test. php:

'components' => [

// ...

'urlManager' => [

'enablePrettyUrl' => true,

'showScriptName' => true,

'rules' => [

['class' => 'yii\rest\UrlRule', 'controller' => 'api/posts'],

],

],

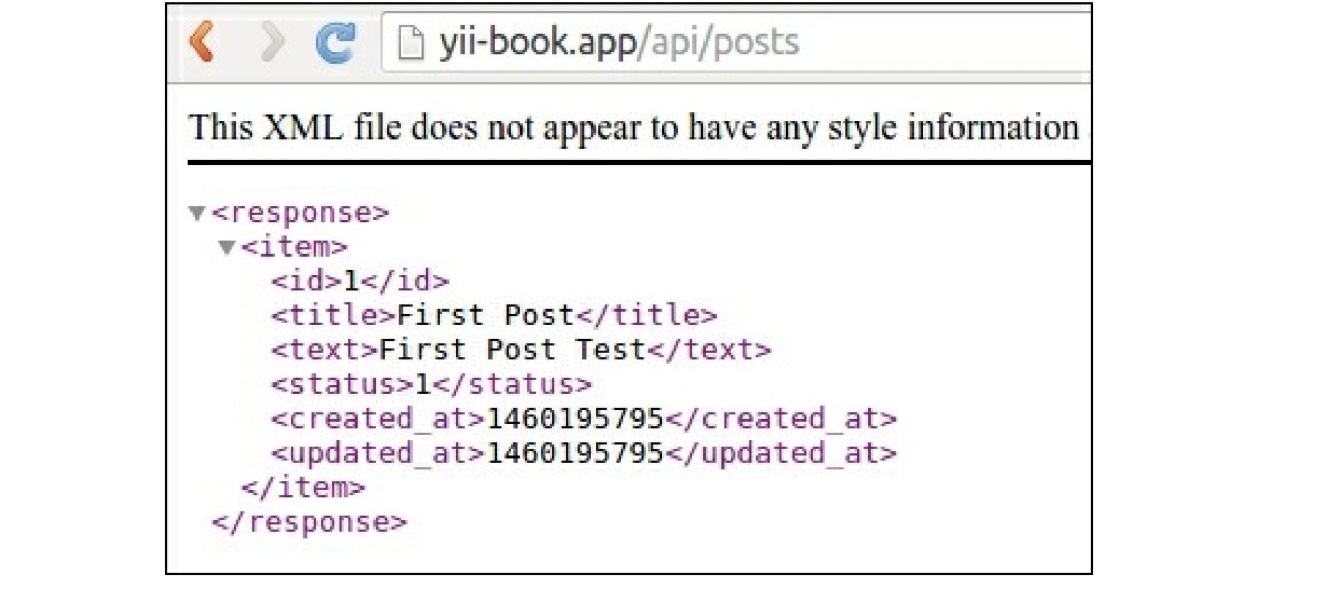
],

1. Add the web/.htaccess file with the following content:

RewriteEngine On

RewriteCond %{REQUEST\_FILENAME} !-f  
RewriteCond %{REQUEST\_FILENAME} !-d  
RewriteRule . index.php

1. Check that the api/posts controller works:



1. Create the API test suite tests/api. suite. yml configuration file with the REST module:

class\_name: ApiTester  
modules:  
enabled:

* REST:

depends: PhpBrowser

url: '<http://127.0.0.1:8080/index-test.php'>

part: [json]

* Yii2:

part: [orm, fixtures]  
entryScript: index-test.php

Now rebuild testers:

composer exec codecept build

1. Create tests/api directory and generate new test class:  
   composer exec codecept generate:cest api Posts
2. Write tests for your REST-API:

<?php

namespace tests\api;  
use ApiTester;

use tests\fixtures\PostFixture;  
use yii\helpers\Url;

class PostsCest  
{

function \_before(ApiTester $I)

{

$I->haveFixtures([

'post' => [

'class' => PostFixture::className(),

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

]

]);

}

public function testGetAll(ApiTester $I)

{

$I->sendGET('/api/posts');

$I->seeResponseCodeIs(200);

$I->seeResponseIsJson();

$I->seeResponseContainsJson([0 => ['title' => 'First Post']]);

}

public function testGetOne(ApiTester $I)

{

$I->sendGET('/api/posts/1');

$I->seeResponseCodeIs(200);

$I->seeResponseIsJson();

$I->seeResponseContainsJson(['title' => 'First Post']);

}

public function testGetNotFound(ApiTester $I)

{

$I->sendGET( '/api/posts/100');

$I->seeResponseCodeIs(404);

$I->seeResponseIsJson();

$I->seeResponseContainsJson(['name' => 'Not Found']);

}

public function testCreate(ApiTester $I)

{

$I->sendPOST('/api/posts', [

'title' => 'Test Title',

'text' => 'Test Text',

]);

$I->seeResponseCodeIs(201);

$I->seeResponseIsJson();

$I->seeResponseContainsJson(['title' => 'Test Title']);

}

public function testUpdate(ApiTester $I)

{

$I->sendPUT('/api/posts/2', [

'title' => 'New Title',

]);

$I->seeResponseCodeIs(200);  
$I->seeResponseIsJson();  
$I->seeResponseContainsJson([

'title' => 'New Title',

'text' => 'Old Text For Updating',

]);

}

public function testDelete(ApiTester $I)

{

$I->sendDELETE('/api/posts/3');  
$I->seeResponseCodeIs(204);

}

}

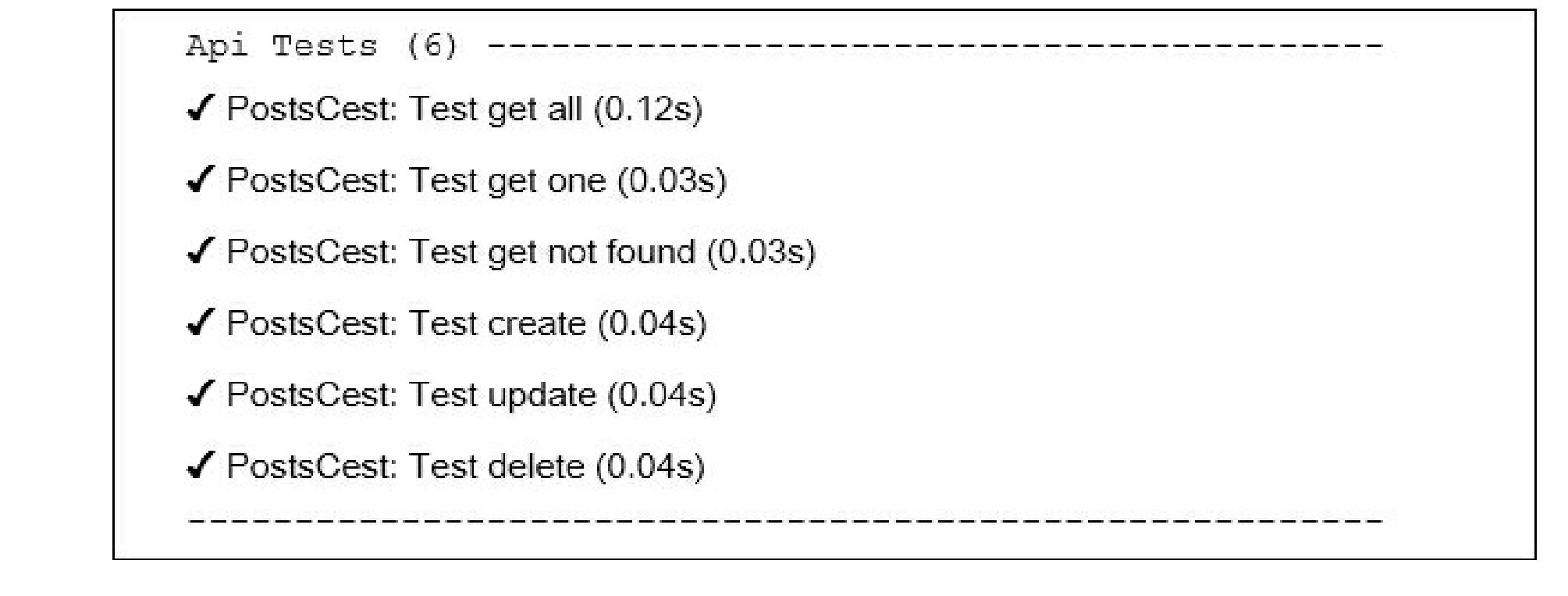
1. Run application server:

tests/bin yii serve

1. Run API tests:

composer exec codecept run api

Now see the result:

All tests passed and our API works correctly.

How it works...

Codeception is high-level testing framework, based on the PHPUnit package for providing infrastructure  
for writing unit, integration, functional, and acceptance tests.

We can use built-in Yii2 module of Codeception which allows us to load fixtures, work with models and  
other things from Yii Framework.

See also

* For further information, refer to:

° [http://c ode c eption. c om/doc s/01-Introduction](http://codeception.com/docs/01-Introduction)  
o [https://phpunit.de/manua]/5.2/en/insta]lat.ion.html](https://phpunit.de/manual/5.2/en/installation.html)

* The tests/README. md file of your basic or advanced application:

° [https://github.com/yiisoft/yii2-app-basic/blob/master/tests/RFADMF.md](https://github.com/yiisoft/yii2-app-basic/blob/master/tests/README.md)  
o [https://github.com/yiisoft/yri2-app-advanced/blob/mastpr/tpsts/RFADME.md](https://github.com/yiisoft/yii2-app-advanced/blob/master/tests/README.md)

* The Unit testing with PHPUnit recipe