docs.zendframework.com

Unit Testing A zend-mvc Application

1. Docs »
2. MVC Tutorials »
3. Unit Testing A zend-mvc Application

A solid unit test suite is essential for ongoing development in large projects, especially those with many people involved. Going back and manually testing every individual component of an application after every change is impractical. Your unit tests will help alleviate that by automatically testing your application's components and alerting you when something is not working the same way it was when you wrote your tests.

This tutorial is written in the hopes of showing how to test different parts of a zend-mvc application. As such, this tutorial will use the application written in the getting started user guide. It is in no way a guide to unit testing in general, but is here only to help overcome the initial hurdles in writing unit tests for zend-mvc applications.

It is recommended to have at least a basic understanding of unit tests, assertions and mocks.

zend-test, which provides testing integration for zend-mvc, uses PHPUnit; this tutorial will cover using that library for testing your applications.

Installing zend-test

zend-test provides PHPUnit integration for zend-mvc, including application scaffolding and custom assertions. You will need to install it:

```
$ composer require --dev zendframework/zend-test
```

The above command will update your `composer.json` file and perform an update for you, which will also setup autoloading rules.

Running the initial tests

Out-of-the-box, the skeleton application provides several tests for the shipped

`Application\Controller\IndexController` class. Now that you have zend-test installed, you can run these:

```
$ ./vendor/bin/phpunit
```

> **PHPUnit invocation on Windows**
>
> On Windows, you need to wrap the command in double quotes:
>
> ```
> $ "vendor/bin/phpunit"
> ```

You should see output similar to the following:

```
 PHPUnit 5.4.6 by Sebastian Bergmann and contributors.

...                                                            3 /
3 (100%)

Time: 116 ms, Memory: 11.00MB

OK (3 tests, 7 assertions)
```

There might be 2 failing tests if you followed the getting started guide. This is because the `Application\IndexController` is overridden by the `AlbumController`. This can be ignored for now.

Now it's time to write our own tests!

Setting up the tests directory

As zend-mvc applications are built from modules that should be standalone blocks of an application, we don't test the application in it's entirety, but module by module.

We will demonstrate setting up the minimum requirements to test a module, the `Album` module we wrote in the user guide, which then can be used as a base for testing any other module.

Start by creating a directory called `test` under `module/Album/` with the following subdirectories:

```
module/
```

```
    Album/
        test/
            Controller/
```

Additionally, add an `autoload-dev` rule in your `composer.json` :

```
"autoload-dev": {
    "psr-4": {
        "ApplicationTest\\": "module/Application/test/",
        "AlbumTest\\": "module/Album/test/"
    }
}
```

When done, run:

```
$ composer dump-autoload
```

The structure of the `test` directory matches exactly with that of the module's source files, and it will allow you to keep your tests well-organized and easy to find.

Bootstrapping your tests

Next, edit the `phpunit.xml.dist` file at the project root; we'll add a new test suite to it. When done, it should read as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit colors="true">
    <testsuites>
        <testsuite name="ZendSkeletonApplication Test Suite">
            <directory>./module/Application/test</directory>
        </testsuite>
        <testsuite name="Album">
            <directory>./module/Album/test</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Now run your new Album test suite from the project root:

```
$ ./vendor/bin/phpunit --testsuite Album
```

> **Windows and PHPUnit**
>
> On Windows, don't forget to wrap the `phpunit` command in double quotes:
>
> ```
> $ "vendor/bin/phpunit" --testsuite Album
> ```

You should get similar output to the following:

```
PHPUnit 5.4.6 by Sebastian Bergmann and contributors.

Time: 0 seconds, Memory: 1.75Mb

No tests executed!
```

Let's write our first test!

Your first controller test

Testing controllers is never an easy task, but the zend-test component makes testing much less cumbersome.

First, create `AlbumControllerTest.php` under `module/Album/test/Controller/` with the following contents:

```
<?php
namespace AlbumTest\Controller;

use Album\Controller\AlbumController;
use Zend\Stdlib\ArrayUtils;
use Zend\Test\PHPUnit\Controller\AbstractHttpControllerTestCase;

class AlbumControllerTest extends AbstractHttpControllerTestCase
{
    protected $traceError = false;

    public function setUp()
    {
```

```
        $configOverrides = [];

        $this->setApplicationConfig(ArrayUtils::merge(

            include __DIR__ . '/../../../../config
/application.config.php',
            $configOverrides
        ));
        parent::setUp();
    }
}
```

The `AbstractHttpControllerTestCase` class we extend here helps us setting up the application itself, helps with dispatching and other tasks that happen during a request, and offers methods for asserting request params, response headers, redirects, and more. See the zend-test documentation for more information.

The principal requirement for any zend-test test case is to set the application config with the `setApplicationConfig()` method. For now, we assume the default application configuration will be appropriate; however, we can override values locally within the test using the `$configOverrides` variable.

Now, add the following method to the `AlbumControllerTest` class:

```
public function testIndexActionCanBeAccessed()
{
    $this->dispatch('/album');
    $this->assertResponseStatusCode(200);
    $this->assertModuleName('Album');
    $this->assertControllerName(AlbumController::class);
    $this->assertControllerClass('AlbumController');
    $this->assertMatchedRouteName('album');
}
```

This test case dispatches the `/album` URL, asserts that the response code is 200, and that we ended up in the desired module and controller.

Assert against controller service names

> For asserting the *controller name* we are using the controller name we defined in our routing configuration for the Album module. In our example this should be defined on line 16 of the `module.config.php` file in the Album module.

If you run:

```
$ ./vendor/bin/phpunit --testsuite Album
```

again, you should see something like the following:

```
PHPUnit 5.4.6 by Sebastian Bergmann and contributors.

.                                                                    1 /
1 (100%)

Time: 124 ms, Memory: 11.50MB

OK (1 test, 5 assertions)
```

A successful first test!

A failing test case

We likely don't want to hit the same database during testing as we use for our web property. Let's add some configuration to the test case to remove the database configuration. In your `AlbumControllerTest::setUp()` method, add the following lines right after the call to `parent::setUp();` :

```
$services = $this->getApplicationServiceLocator();
$config = $services->get('config');
unset($config['db']);
$services->setAllowOverride(true);
$services->setService('config', $config);
$services->setAllowOverride(false);
```

The above removes the 'db' configuration entirely; we'll be replacing it with something else before long.

When we run the tests now:

```
 $ ./vendor/bin/phpunit --testsuite Album
PHPUnit 5.4.6 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 8.50Mb

There was 1 failure:

1) AlbumTest\Controller
\AlbumControllerTest::testIndexActionCanBeAccessed
Failed asserting response code "200", actual status code is "500"

{projectPath}/vendor/zendframework/zend-test/src/PHPUnit/Controller
/AbstractControllerTestCase.php:{lineNumber}
{projectPath}/module/Album/test/AlbumTest/Controller
/AlbumControllerTest.php:{lineNumber}

FAILURES!
Tests: 1, Assertions: 0, Failures: 1.
```

The failure message doesn't tell us much, apart from that the expected status code is not 200, but 500. To get a bit more information when something goes wrong in a test case, we set the protected `$traceError` member to `true` (which is the default; we set it to `false` to demonstrate this capability). Modify the following line from just above the `setUp` method in our `AlbumControllerTest` class:

```
 protected $traceError = true;
```

Running the `phpunit` command again and we should see some more information about what went wrong in our test. You'll get a list of the exceptions raised, along with their messages, the filename, and line number:

```
 1) AlbumTest\Controller
\AlbumControllerTest::testIndexActionCanBeAccessed
Failed asserting response code "200", actual status code is "500"

Exceptions raised:
Exception 'Zend\ServiceManager\Exception\ServiceNotCreatedException'
with message 'Service with name "Zend\Db\Adapter\AdapterInterface"
could not be created. Reason: createDriver expects a "driver" key to be
present inside the parameters' in {projectPath}/vendor/zendframework
```

```
/zend-servicemanager/src/ServiceManager.php:{lineNumber}

Exception 'Zend\Db\Adapter\Exception\InvalidArgumentException' with
message 'createDriver expects a "driver" key to be present inside the
parameters' in {projectPath}/vendor/zendframework/zend-db/src/Adapter
/Adapter.php:{lineNumber}
```

Based on the exception messages, it appears we are unable to create a zend-db adapter instance, due to missing configuration!

Configuring the service manager for the tests

The error says that the service manager can not create an instance of a database adapter for us. The database adapter is indirectly used by our `Album\Model\AlbumTable` to fetch the list of albums from the database.

The first thought would be to create an instance of an adapter, pass it to the service manager, and let the code run from there as is. The problem with this approach is that we would end up with our test cases actually doing queries against the database. To keep our tests fast, and to reduce the number of possible failure points in our tests, this should be avoided.

The second thought would be then to create a mock of the database adapter, and prevent the actual database calls by mocking them out. This is a much better approach, but creating the adapter mock is tedious (but no doubt we will have to create it at some point).

The best thing to do would be to mock out our `Album\Model\AlbumTable` class which retrieves the list of albums from the database. Remember, we are now testing our controller, so we can mock out the actual call to `fetchAll` and replace the return values with dummy values. At this point, we are not interested in how `fetchAll()` retrieves the albums, but only that it gets called and that it returns an array of albums; these facts allow us to provide mock instances. When we test `AlbumTable` itself, we can write the actual tests for the `fetchAll` method.

First, let's do some setup.

Add import statements to the top of the test class file for each of the `AlbumTable` and `ServiceManager` classes:

```
 use Album\Model\AlbumTable;
use Zend\ServiceManager\ServiceManager;
```

Now add the following property to the test class:

```
protected $albumTable;
```

Next, we'll create three new methods that we'll invoke during setup:

```
protected function configureServiceManager(ServiceManager $services)
{
    $services->setAllowOverride(true);

    $services->setService('config',
$this->updateConfig($services->get('config')));
    $services->setService(AlbumTable::class,
$this->mockAlbumTable()->reveal());

    $services->setAllowOverride(false);
}

protected function updateConfig($config)
{
    $config['db'] = [];
    return $config;
}

protected function mockAlbumTable()
{
    $this->albumTable = $this->prophesize(AlbumTable::class);
    return $this->albumTable;
}
```

By default, the `ServiceManager` does not allow us to replace existing services. `configureServiceManager()` calls a special method on the instance to enable overriding services, and then we inject specific overrides we wish to use. When done, we disable overrides to ensure that if, during dispatch, any code attempts to override a service, an exception will be raised.

The last method above creates a mock instance of our `AlbumTable` using Prophecy, an object mocking framework that's bundled and integrated in PHPUnit. The instance returned by `prophesize()` is a scaffold object; calling `reveal()` on it, as done in the `configureServiceManager()` method above, provides the underlying mock object that will then be asserted against.

With this in place, we can update our `setUp()` method to read as follows:

```
public function setUp()
{




    $configOverrides = [];

    $this->setApplicationConfig(ArrayUtils::merge(
        include __DIR__ . '/../../../../config/application.config.php',
        $configOverrides
    ));

    parent::setUp();


$this->configureServiceManager($this->getApplicationServiceLocator());
}
```

Now update the `testIndexActionCanBeAccessed()` method to add a line asserting the `AlbumTable` 's `fetchAll()` method will be called, and return an array:

```
public function testIndexActionCanBeAccessed()
{
    $this->albumTable->fetchAll()->willReturn([]);

    $this->dispatch('/album');
    $this->assertResponseStatusCode(200);
    $this->assertModuleName('Album');
    $this->assertControllerName(AlbumController::class);
    $this->assertControllerClass('AlbumController');
    $this->assertMatchedRouteName('album');
}
```

Running `phpunit` at this point, we will get the following output as the tests now pass:

```
$ ./vendor/bin/phpunit --testsuite Album
PHPUnit 5.4.6 by Sebastian Bergmann and contributors.
```

```
.                                                                1 /
1 (100%)

Time: 105 ms, Memory: 10.75MB

OK (1 test, 5 assertions)
```

Testing actions with POST

A common scenario with controllers is processing POST data submitted via a form, as we do in the `AlbumController::addAction()`. Let's write a test for that.

```
public function testAddActionRedirectsAfterValidPost()
{
    $this->albumTable
        ->saveAlbum(Argument::type(Album::class))
        ->shouldBeCalled();

    $postData = [
        'title'  => 'Led Zeppelin III',
        'artist' => 'Led Zeppelin',
        'id'     => '',
    ];
    $this->dispatch('/album/add', 'POST', $postData);
    $this->assertResponseStatusCode(302);
    $this->assertRedirectTo('/album');
}
```

This test case references two new classes that we need to import; add the following import statements at the top of the class file:

```
use Album\Model\Album;
use Prophecy\Argument;
```

`Prophecy\Argument` allows us to perform assertions against the values passed as arguments to mock objects. In this case, we want to assert that we received an `Album` instance. (We could have also done deeper assertions to ensure the `Album` instance contained expected data.)

When we dispatch the application this time, we use the request method POST, and pass data to it. This test case then asserts a 302 response status, and introduces a new assertion against the location to which the response redirects.

Running `phpunit` gives us the following output:

```
 $ ./vendor/bin/phpunit --testsuite Album
PHPUnit 5.4.6 by Sebastian Bergmann and contributors.

..                                                                  2 /
2 (100%)

Time: 1.49 seconds, Memory: 13.25MB

OK (2 tests, 8 assertions)
```

Testing the `editAction()` and `deleteAction()` methods can be performed similarly; however, when testing the `editAction()` method, you will also need to assert against the `AlbumTable::getAlbum()` method:

```
 $this->albumTable->getAlbum($id)->willReturn(new Album());
```

Ideally, you should test all the various paths through each method. For example:

- Test that a non-POST request to `addAction()` displays an empty form.
- Test that a invalid data provided to `addAction()` re-displays the form, but with error messages.
- Test that absence of an identifier in the route parameters when invoking either `editAction()` or `deleteAction()` will redirect to the appropriate location.
- Test that an invalid identifier passed to `editAction()` will redirect to the album landing page.
- Test that non-POST requests to `editAction()` and `deleteAction()` display forms.

and so on. Doing so will help you understand the paths through your application and controllers, as well as ensure that changes in behavior bubble up as test failures.

Testing model entities

Now that we know how to test our controllers, let us move to an other important part of our application: the model entity.

Here we want to test that the initial state of the entity is what we expect it to be, that we can convert the model's parameters to and from an array, and that it has all the input filters we need.

Create the file `AlbumTest.php` in `module/Album/test/Model` directory with the following contents:

```
 <?php
namespace AlbumTest\Model;

use Album\Model\Album;
use PHPUnit_Framework_TestCase as TestCase;

class AlbumTest extends TestCase
{
    public function testInitialAlbumValuesAreNull()
    {
        $album = new Album();

        $this->assertNull($album->artist, '"artist" should be null by
default');
        $this->assertNull($album->id, '"id" should be null by default');
        $this->assertNull($album->title, '"title" should be null by
default');
    }

    public function testExchangeArraySetsPropertiesCorrectly()
    {
        $album = new Album();
        $data  = [
            'artist' => 'some artist',
            'id'     => 123,
            'title'  => 'some title'
        ];

        $album->exchangeArray($data);

        $this->assertSame(
            $data['artist'],
            $album->artist,
            '"artist" was not set correctly'
        );

        $this->assertSame(
            $data['id'],
            $album->id,
            '"id" was not set correctly'
        );

        $this->assertSame(
            $data['title'],
            $album->title,
            '"title" was not set correctly'
        );
    }

    public function
testExchangeArraySetsPropertiesToNullIfKeysAreNotPresent()
    {
        $album = new Album();

        $album->exchangeArray([
            'artist' => 'some artist',
            'id'     => 123,
            'title'  => 'some title',
        ]);
        $album->exchangeArray([]);

        $this->assertNull($album->artist, '"artist" should default to
null');
        $this->assertNull($album->id, '"id" should default to null');
        $this->assertNull($album->title, '"title" should default to
null');
    }

    public function testGetArrayCopyReturnsAnArrayWithPropertyValues()
    {
        $album = new Album();
        $data  = [
            'artist' => 'some artist',
            'id'     => 123,
            'title'  => 'some title'
        ];

        $album->exchangeArray($data);
        $copyArray = $album->getArrayCopy();

        $this->assertSame($data['artist'], $copyArray['artist'],
'"artist" was not set correctly');
        $this->assertSame($data['id'], $copyArray['id'], '"id" was not
set correctly');
        $this->assertSame($data['title'], $copyArray['title'], '"title"
was not set correctly');
    }

    public function testInputFiltersAreSetCorrectly()
    {
        $album = new Album();

        $inputFilter = $album->getInputFilter();

        $this->assertSame(3, $inputFilter->count());
        $this->assertTrue($inputFilter->has('artist'));
        $this->assertTrue($inputFilter->has('id'));
        $this->assertTrue($inputFilter->has('title'));
    }
}
```

We are testing for 5 things:

1. Are all of the `Album` 's properties initially set to `NULL` ?
2. Will the `Album` 's properties be set correctly when we call `exchangeArray()` ?
3. Will a default value of `NULL` be used for properties whose keys are not present in the `$data` array?
4. Can we get an array copy of our model?
5. Do all elements have input filters present?

If we run `phpunit` again, we will get the following output, confirming that our model is indeed correct:

```
 $ ./vendor/bin/phpunit --testsuite Album
PHPUnit 5.4.6 by Sebastian Bergmann and contributors.

.......                                                            7 /
7 (100%)

Time: 186 ms, Memory: 13.75MB

OK (7 tests, 24 assertions)
```

Testing model tables

The final step in this unit testing tutorial for zend-mvc applications is writing tests for our model tables.

This test assures that we can get a list of albums, or one album by its ID, and that we can save and delete albums from the database.

To avoid actual interaction with the database itself, we will replace certain parts with mocks.

Create a file `AlbumTableTest.php` in `module/Album/test/Model/` with the following contents:

```
 <?php
namespace AlbumTest\Model;

use Album\Model\AlbumTable;
use Album\Model\Album;
use PHPUnit_Framework_TestCase as TestCase;
use RuntimeException;
use Zend\Db\ResultSet\ResultSetInterface;
use Zend\Db\TableGateway\TableGatewayInterface;

class AlbumTableTest extends TestCase
{
    protected function setUp()
    {
        $this->tableGateway =
$this->prophesize(TableGatewayInterface::class);
        $this->albumTable = new
AlbumTable($this->tableGateway->reveal());
```

```
    }

    public function testFetchAllReturnsAllAlbums()
    {
        $resultSet =
$this->prophesize(ResultSetInterface::class)->reveal();
        $this->tableGateway->select()->willReturn($resultSet);

        $this->assertSame($resultSet, $this->albumTable->fetchAll());
    }
}
```

Since we are testing the `AlbumTable` here and not the `TableGateway` class (which has already been tested in zend-db), we only want to make sure that our `AlbumTable` class is interacting with the `TableGateway` class the way that we expect it to. Above, we're testing to see if the `fetchAll()` method of `AlbumTable` will call the `select()` method of the `$tableGateway` property with no parameters. If it does, it should return a `ResultSet` instance. Finally, we expect that this same `ResultSet` object will be returned to the calling method. This test should run fine, so now we can add the rest of the test methods:

```
 public function testCanDeleteAnAlbumByItsId()
{
    $this->tableGateway->delete(['id' => 123])->shouldBeCalled();
    $this->albumTable->deleteAlbum(123);
}

public function
testSaveAlbumWillInsertNewAlbumsIfTheyDontAlreadyHaveAnId()
{
    $albumData = [
        'artist' => 'The Military Wives',
        'title'  => 'In My Dreams'
    ];
    $album = new Album();
    $album->exchangeArray($albumData);

    $this->tableGateway->insert($albumData)->shouldBeCalled();
    $this->albumTable->saveAlbum($album);
}

public function
```

```
testSaveAlbumWillUpdateExistingAlbumsIfTheyAlreadyHaveAnId()
{
    $albumData = [
        'id'     => 123,
        'artist' => 'The Military Wives',
        'title'  => 'In My Dreams',
    ];
    $album = new Album();
    $album->exchangeArray($albumData);

    $resultSet = $this->prophesize(ResultSetInterface::class);
    $resultSet->current()->willReturn($album);

    $this->tableGateway
        ->select(['id' => 123])
        ->willReturn($resultSet->reveal());
    $this->tableGateway
        ->update(
            array_filter($albumData, function ($key) {
                return in_array($key, ['artist', 'title']);
            }, ARRAY_FILTER_USE_KEY),
            ['id' => 123]
        )->shouldBeCalled();

    $this->albumTable->saveAlbum($album);
}

public function testExceptionIsThrownWhenGettingNonExistentAlbum()
{
    $resultSet = $this->prophesize(ResultSetInterface::class);
    $resultSet->current()->willReturn(null);

    $this->tableGateway
        ->select(['id' => 123])
        ->willReturn($resultSet->reveal());

    $this->setExpectedException(
        RuntimeException::class,
        'Could not find row with identifier 123'
    );
    $this->albumTable->getAlbum(123);
}
```

These tests are nothing complicated and should be self explanatory. In each test, we add assertions to our mock table gateway, and then call and assert against methods in our `AlbumTable`.

We are testing that:

1. We can retrieve an individual album by its ID.
2. We can delete albums.
3. We can save a new album.
4. We can update existing albums.
5. We will encounter an exception if we're trying to retrieve an album that doesn't exist.

Running `phpunit` one last time, we get the output as follows:

```
 $ ./vendor/bin/phpunit --testsuite Album
PHPUnit 5.4.6 by Sebastian Bergmann and contributors.

.............                                                    13 /
13 (100%)

Time: 151 ms, Memory: 14.00MB

OK (13 tests, 31 assertions)
```

Conclusion

In this short tutorial, we gave a few examples how different parts of a zend-mvc application can be tested. We covered setting up the environment for testing, how to test controllers and actions, how to approach failing test cases, how to configure the service manager, as well as how to test model entities and model tables.

This tutorial is by no means a definitive guide to writing unit tests, just a small stepping stone helping you develop applications of higher quality.