Testing WebServices

The same way we tested a web site, Codeception allows you to test web services. They are very hard to test manually, so it's a really good idea to automate web service testing. We have SOAP and REST as standards, which are represented in corresponding modules, which we will cover in this chapter.

You should start by creating a new test suite, (which was not provided by the `bootstrap` command). We recommend calling it **api** and using the `ApiTester` class for it.

```
$ php codecept generate:suite api
```

We will put all the api tests there.

REST

The REST web service is accessed via HTTP with standard methods: `GET`, `POST`, `PUT`, `DELETE`. They allow users to receive and manipulate entities from the service. Accessing a WebService requires an HTTP client, so for using it you need the module `PhpBrowser` or one of framework modules set up. For example, we can use the `Symfony` module for Symfony2 applications in order to ignore web server and test web service internally.

Configure modules in `api.suite.yml`:

```
actor: ApiTester
modules:
    enabled:
        - REST:
            url: http://serviceapp/api/v1/
            depends: PhpBrowser
```

The REST module will connect to `PhpBrowser` according to this configuration. Depending on the web service we may deal with XML or JSON responses. Codeception handles both data formats well, however If you don't need one of them, you can explicitly specify that the JSON or XML parts of the module will be used:

```
actor: ApiTester
modules:
    enabled:
        - REST:
            url: http://serviceapp/api/v1/
            depends: PhpBrowser
            part: Json
```

API tests can be functional and be executed using Symfony, Laravel5, Zend, or any other framework module. You will need slightly update configuration for it:

```
actor: ApiTester
modules:
    enabled:
        - REST:
            url: /api/v1/
            depends: Laravel5
```

Once we have configured our new testing suite, we can create the first sample test:

```
$ php codecept generate:cept api CreateUser
```

It will be called `CreateUserCept.php` . We can use it to test the creation of a user via the REST API.

```
<?php
$I = new ApiTester($scenario);
$I->wantTo('create a user via API');
$I->amHttpAuthenticated('service_user', '123456');
$I->haveHttpHeader('Content-Type', 'application/x-www-form-urlencoded');
$I->sendPOST('/users', ['name' => 'davert', 'email' => 'davert@codeception.com']);
$I->seeResponseCodeIs(\Codeception\Util\HttpCode::OK); // 200
$I->seeResponseIsJson();
$I->seeResponseContains('{"result":"ok"}');
```

We can use HTTP code constants from `Codeception\Util\HttpCode` instead of numeric values to check response code in `seeResponseCodeIs` and `dontSeeResponseCodeIs` methods.

Testing JSON Responses

The last line of the previous example verified that the response contained the provided string. However we shouldn't rely on it, as depending on content formatting we can receive different results with the same data. What we actually need is to check that the response can be parsed and it contains some of the values we expect. In the case of JSON we can use the `seeResponseContainsJson` method

```
<?php
// matches {"result":"ok"}'
$I->seeResponseContainsJson(['result' => 'ok']);
// it can match tree-like structures as well
$I->seeResponseContainsJson([
  'user' => [
      'name' => 'davert',
      'email' => 'davert@codeception.com',
      'status' => 'inactive'
  ]
]);
```

You may want to perform even more complex assertions on a response. This can be done by writing your own methods in the Helper (http://codeception.com/docs/06-ReusingTestCode#Modules-and-Helpers) classes. To access the latest JSON response you will need to get the `response` property of the `REST` module. Let's demonstrate it with the `seeResponseIsHtml` method:

```
<?php
namespace Helper;

class Api extends \Codeception\Module
{
  public function seeResponseIsHtml()
  {
    $response = $this->getModule('REST')->response;
    $this->assertRegExp('~^<!DOCTYPE HTML(.*?)<html>.*?<\/html>~m', $response);
  }
}
```

The same way you can receive request parameters and headers.

Validate JSON structures

It is pretty common for API tests to not only validate the received data but to check the structure of the response. Response data is not usually considered to be consistent, and may change on each request, however the JSON/XML structure should be kept the same for an API version. In order to check response structure the REST module has some useful methods.

If we expect a JSON response to be received we can check its structure with JSONPath (http://goessner.net/articles/JsonPath/). It looks and sounds like XPath but is designed to work with JSON data, however we can convert JSON into XML and use XPath to validate the structure. Both approaches are valid and can be used in the REST module:

```
<?php
$I = new ApiTester($scenario);
$I->wantTo('validate structure of GitHub api responses');
$I->sendGET('/users');
$I->seeResponseCodeIs(HttpCode::OK); // 200
$I->seeResponseIsJson();
$I->seeResponseJsonMatchesJsonPath('$[0].user.login');
$I->seeResponseJsonMatchesXpath('//user/login');
```

More detailed check can be applied if you need to validate the type of fields in a response. You can do that by using with a seeResponseMatchesJsonType (http://codeception.com/docs/modules/REST#seeResponseMatchesJsonType) action in which you define the structure of JSON response.

```
<?php
$I->sendGET('/users/1');
$I->seeResponseCodeIs(HttpCode::OK); // 200
$I->seeResponseIsJson();
$I->seeResponseMatchesJsonType([
    'id' => 'integer',
    'name' => 'string',
    'email' => 'string:email',
    'homepage' => 'string:url|null',
    'created_at' => 'string:date',
    'is_active' => 'boolean'
]);
```

Codeception uses this simple and lightweight definitions format which can be easily learned and extended (http://codeception.com/docs/modules/REST#seeResponseMatchesJsonType).

Testing XML Responses

In case your REST API works with XML format you can use similar methods to test its data and structure. There is `seeXmlResponseIncludes` method to match inclusion of XML parts in response, and `seeXmlResponseMatchesXpath` to validate its structure.

```
<?php
use Codeception\Util\Xml as XmlUtils;

$I = new ApiTester($scenario);
$I->wantTo('validate structure of GitHub api responses');
$I->sendGET('/users.xml');
$I->seeResponseCodeIs(\Codeception\Util\HttpCode::OK); // 200
$I->seeResponseIsXml();
$I->seeXmlResponseMatchesXpath('//user/login');
$I->seeXmlResponseIncludes(XmlUtils::toXml(
    'user' => [
      'name' => 'davert',
      'email' => 'davert@codeception.com',
      'status' => 'inactive'
  ]
));
```

We are using XmlUtils class which allows us to build XML structures in a clean manner. The `toXml` method may accept a string or array and returns \DOMDocument instance. If your XML contains attributes and so can't be represented as a PHP array you can create XML using the XmlBuilder (http://codeception.com/docs/reference/XmlBuilder) class. We will take a look at it a bit more in next section.

Use `\Codeception\Util\Xml::build()` to create XmlBuilder instance.

SOAP

SOAP web services are usually more complex. You will need PHP configured with SOAP support (http://php.net/manual/en/soap.installation.php). Good knowledge of XML is required too. `SOAP` module uses specially formatted POST request to connect to WSDL web services. Codeception uses `PhpBrowser` or one of framework modules to perform interactions. If you choose using a framework module, SOAP will automatically connect to the underlying framework. That may improve the speed of a test execution and will provide you with more detailed stack traces.

Let's configure `SOAP` module to be used with `PhpBrowser`:

```
actor: ApiTester
modules:
    enabled:
    - SOAP:
      depends: PhpBrowser
      endpoint: http://serviceapp/api/v1/
```

SOAP request may contain application specific information, like authentication or payment. This information is provided with SOAP header inside the `<soap:Header>` element of XML request. In case you need to submit such header, you can use `haveSoapHeader` action. For example, next line of code

```
<?php
$I->haveSoapHeader('Auth', array('username' => 'Miles', 'password' => '123456'));
```

will produce this XML header

```
<soap:Header>
<Auth>
  <username>Miles</username>
  <password>123456</password>
</Auth>
</soap:Header>
```

Use `sendSoapRequest` method to define the body of your request.

```
<?php
$I->sendSoapRequest('CreateUser', '<name>Miles Davis</name><email>miles@davis.com</email>');
```

This call will be translated to XML:

```
<soap:Body>
<ns:CreateUser>
  <name>Miles Davis</name>
  <email>miles@davis.com</email>
</ns:CreateUser>
</soap:Body>
```

And here is the list of sample assertions that can be used with SOAP.

```
<?php
$I->seeSoapResponseEquals('<?xml version="1.0"<error>500</error>');
$I->seeSoapResponseIncludes('<result>1</result>');
$I->seeSoapResponseContainsStructure('<user><name></name><email></email>');
$I->seeSoapResponseContainsXPath('//result/user/name[@id=1]');
```

In case you don't want to write long XML strings, consider using XmlBuilder (http://codeception.com/docs/reference/XmlBuilder) class. It will help you to build complex XMLs in jQuery-like style. In the next example we will use `XmlBuilder` instead of regular XML.

```
<?php
use \Codeception\Util\Xml;

$I = new ApiTester($scenario);
$I->wantTo('create user');
$I->haveSoapHeader('Session', array('token' => '123456'));
$I->sendSoapRequest('CreateUser', Xml::build()
  ->user->email->val('miles@davis.com'));
$I->seeSoapResponseIncludes(Xml::build()
  ->result->val('Ok')
    ->user->attr('id', 1)
);
```

It's up to you to decide whether to use `XmlBuilder` or plain XML. `XmlBuilder` will return XML string as well.

You may extend current functionality by using `SOAP` module in your helper class. To access the SOAP response as `\DOMDocument` you can use `response` property of `SOAP` module.

```
<?php
namespace Helper;
class Api extends \Codeception\Module {

  public function seeResponseIsValidOnSchema($schema)
  {
    $response = $this->getModule('SOAP')->response;
    $this->assertTrue($response->schemaValidate($schema));
  }
}
```

Conclusion

Codeception has two modules that will help you to test various web services. They need a new `api` suite to be created. Remember, you are not limited to test only response body. By including `Db` module you may check if a user has been created after the `CreateUser` call. You can improve testing scenarios by using REST or SOAP responses in your helper methods.

- **Next Chapter: Codecoverage > (/docs/11-Codecoverage)**
- **Previous Chapter: < Data (/docs/09-Data)**

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)

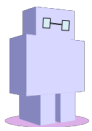
Thanks to


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


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


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

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.