

Using Guzzle and PHPUnit for REST API Testing

28 Dec 2016 by *Junade Ali*.



APIs are increasingly becoming the backbone of the modern internet - whether you're ordering food from an app on your phone or browsing a blog using a modern JavaScript framework, chances are those requests are flowing through an API. Given the need for APIs to evolve through refactoring and extension, having great automated tests allows you to develop fast without needing to slow down to run manual tests to work out what's broken. Additionally, by having tests in place you're able to firmly identify the requirements that your API should meet, your API tests effectively form a tangible and executable specification. API Testing offers an end-to-end mechanism of testing the behaviour of your API which has advantages in both reliability and also development productivity.

In this post I'll be demonstrating how you can test RESTful APIs in an automated fashion using PHP, by building a testing framework through creative use of two packages - Guzzle and PHPUnit. The resulting tests will be something you can run outside of your API as part of your deployment or CI (Continuous Integration) process.

Guzzle acts as a powerful HTTP client which we can use to simulate HTTP Requests against our API. Though PHPUnit acts as a Unit Test framework (based on XUnit), in this case we will be using this powerful testing framework to test the HTTP responses we get back from our APIs using Guzzle.

Preparing our Environment

In order to pull in the required packages, we'll be using Composer - a dependency manager for PHP. Inside our Composer project, we can simply require the dependencies we're after:

```
$ composer require phpunit/phpunit
$ composer require guzzlehttp/guzzle
$ composer update
```

When we ran `composer require` for each of the two packages, Composer went ahead and actually downloaded the packages we want, these are stored in the `vendor` directory. Additionally when we ran `composer update`, Composer updated its PSR-4 (<http://www.php-fig.org/psr/psr-4/>) autoload script that allows us to pull in all the dependencies we've required with one file include, you can find this in `vendor/autoload.php`.

With our dependencies in place, we can now configure PHPUnit to use Guzzle. In order to do this, we need to tell PHPUnit where our Composer autoload file is, but also where our tests are located. We can do this through writing a `phpunit.xml` in the root directory of our project:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="vendor/autoload.php">
  <testsuites>
    <testsuite name="REST API Test Suite">
      <directory suffix=".php">./tests/</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

In the XML above, the two noteworthy elements are the opening `phpunit` tag (which defines with a `bootstrap` property where our Composer autoload script is), additionally we have a `testsuite` element which defines our test suites (and a child `directory` property to define where the specific tests live). From here, we can just add an empty directory called `tests` for our tests to reside in.

If we now run PHPUnit (through the command `./vendor/bin/phpunit`), we should see an output similar to the one I get below:

```
Test-the-REST — -bash — 65x20
Junades-MacBook-Pro:Test-the-REST junade$ ./vendor/bin/phpunit
PHPUnit 5.7.4 by Sebastian Bergmann and contributors.

Time: 32 ms, Memory: 2.00MB

No tests executed!
Junades-MacBook-Pro:Test-the-REST junade$
```

With our environment defined, we're ready to move on to the next step. First; purely for the sake of convenience, I've added a shortcut to my `composer.json` file so that when I run `composer test` it will point to `./vendor/bin/phpunit`. You can do this by adding the following JSON to your `composer.json` file:

```
"scripts": {
  "test": "./vendor/bin/phpunit"
}
```

Writing our Tests

As an example, I'll be writing tests against an endpoint at `httpbin.org`. The first test I'll write will be against the `/user-agent` endpoint, so I'll create a file called `UserAgentTest.php`, be sure to extend the `PHPUnit_Framework_TestCase` class:

```
<?php

class UserAgentTest extends PHPUnit_Framework_TestCase
{
}
```

Before each test, PHPUnit will run the `setUp` method and after the test has executed it will run the `tearDown` method in the class (if they exist). By utilising these methods we can instantiate our Guzzle HTTP client ready for each test and then return to a clean slate afterwards:

```
<?php

class UserAgentTest extends PHPUnit_Framework_TestCase
{
    private $http;

    public function setUp()
    {
        $this->http = new GuzzleHttp\Client(['base_uri' => 'https://httpbin.org/']);
    }

    public function tearDown() {
        $this->http = null;
    }
}
```

Note that if you feel even more adventurous, you can utilise environment variables (through the `getenv` method) to set the `baseurl` - for this tutorial however, I'll be keeping things simple.

With our `setUp` and `tearDown` methods in place, we can now go ahead and actually create our test methods. As I'll start off by testing against the `GET` HTTP verb, I'll name the first test method `testGet`. From here, we can make the request and then check the properties we get back.

```
public function testGet()
{
    $response = $this->http->request('GET', 'user-agent');

    $this->assertEquals(200, $response->getStatusCode());

    $contentType = $response->getHeaders()['Content-Type'][0];
    $this->assertEquals("application/json", $contentType);

    $userAgent = json_decode($response->getBody())->{"user-agent"};
    $this->assertRegExp('/Guzzle/', $userAgent);
}
```

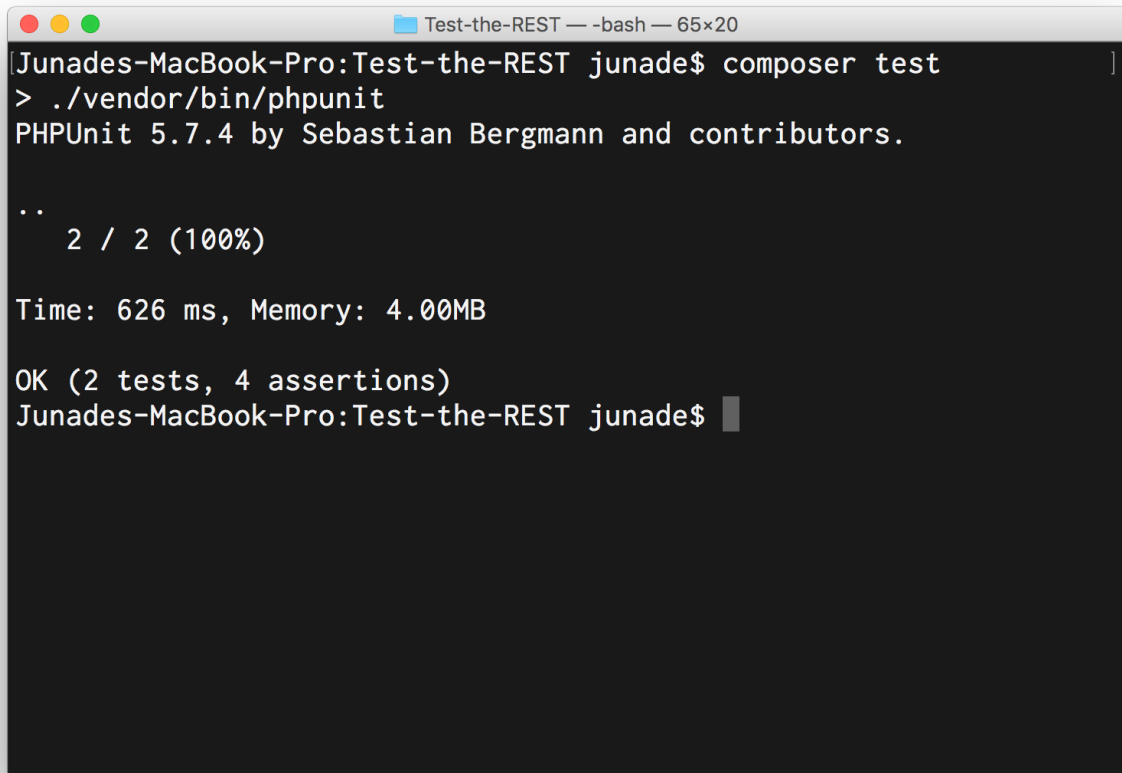
In the method above, I've made a GET request to the user-agent endpoint. I can then check the response code I get back was indeed `200` using the first assertion. The next assertion I test against is whether the `Content-Type` header indicates the response is JSON. Finally I check that the JSON body itself actually contains the phrase "Guzzle" in the user-agent property.

We can add additional assertions as required, but we can also add additional methods for other HTTP verbs. For example, here's a simple test to see if I get a `405` status code when I make a PUT request to the `/user-agent` endpoint:

```
public function testPut()
{
    $response = $this->http->request('PUT', 'user-agent', ['http_errors' => false]);

    $this->assertEquals(405, $response->getStatusCode());
}
```

Next time we run PHPUnit, we can see if our tests pass successfully and also get insight into some statistics surrounding the execution of these tests:

A terminal window titled "Test-the-REST — -bash — 65x20" showing the execution of PHPUnit tests. The user runs "composer test" and then "./vendor/bin/phpunit". The output shows two tests passing with 100% success rate, execution time of 626 ms, and memory usage of 4.00MB. The tests are OK with 2 tests and 4 assertions.

```
Junades-MacBook-Pro:Test-the-REST junade$ composer test
> ./vendor/bin/phpunit
PHPUnit 5.7.4 by Sebastian Bergmann and contributors.

..
 2 / 2 (100%)

Time: 626 ms, Memory: 4.00MB

OK (2 tests, 4 assertions)
Junades-MacBook-Pro:Test-the-REST junade$
```

Conclusion

That's all there is to this simple approach to API testing. If you want some insight into the overall code, feel free to review the project files in the [Github repository \(https://github.com/IcyApril/Test-the-REST\)](https://github.com/IcyApril/Test-the-REST).

If you find yourself using this testing set-up, be sure to review the [Guzzle Request Options \(http://docs.guzzlephp.org/en/latest/request-options.html\)](http://docs.guzzlephp.org/en/latest/request-options.html) to learn what kind of HTTP requests you can run with Guzzle and also check out the [types of assertions \(https://phpunit.de/manual/current/en/appendixes.assertions.html\)](https://phpunit.de/manual/current/en/appendixes.assertions.html) of you can run with PHPUnit.

comments powered by Disqus