BY OLUYEMI OLUSUSI ▪ 2020-06-05

Getting Started with Unit Testing a Laravel API using PHPUnit

Performing unit, automated feature, and API endpoint testings are considered as some of the best practices to ensure proper implementation of specified software requirements, because they help guarantee the success of such applications. Testing, by all means, tends to give you a 100 percent assurance that any incremental changes and newly implemented features in your project won't break the app. This practice is often referred to as Test-driven Development.

Laravel, as one of the popular PHP frameworks was built with testing in mind and comes with a testing suite named PHPUnit. PHPUnit is a testing framework built to enhance PHP developers' productivity during development. It is primarily designed for testing PHP code in the smallest possible components known as unit testing, but also flexible enough to be used beyond unit testing.

In this tutorial, we will take a test-driven development approach and learn how to test the endpoints of a Laravel API project. We will start by writing tests, expecting them to fail. Afterward, we will write the code to make our tests pass. By the time we are done, you will have learned how to carry out basic testing and be confident enough to apply this knowledge on your new or existing Laravel API projects.

Prerequisites

Basic knowledge of building applications with Laravel will be of help in this tutorial. Also, you need to ensure that you have installed Composer globally to manage dependencies.

Getting Started

Our Laravel API will be used to create a list and display details of top tech CEOs in the world. This is similar to what we built in a previous post. To get started as quickly as possible, download the starter project that contains the structures that enable our application to work as specified.

To begin, run the following command to download the starter project using Git:

```
1 $ git clone https://github.com/yemiwebby/laravel-api-testing-starter.g
```

Next, move into the new project's folder and install all its dependencies:

```
// move into the new folder
$ cd laravel-api-testing-starter

//install dependencies
$ composer install
```

This sample project already contains the following:

- *User* and *CEO* Models with respective fillable properties
- Controllers, migration files, as well as an API route contained in the `routes/api.php` file. And finally,
- Laravel Passport installed and configured to work with the project. Read this article to learn more about securing the Laravel API.

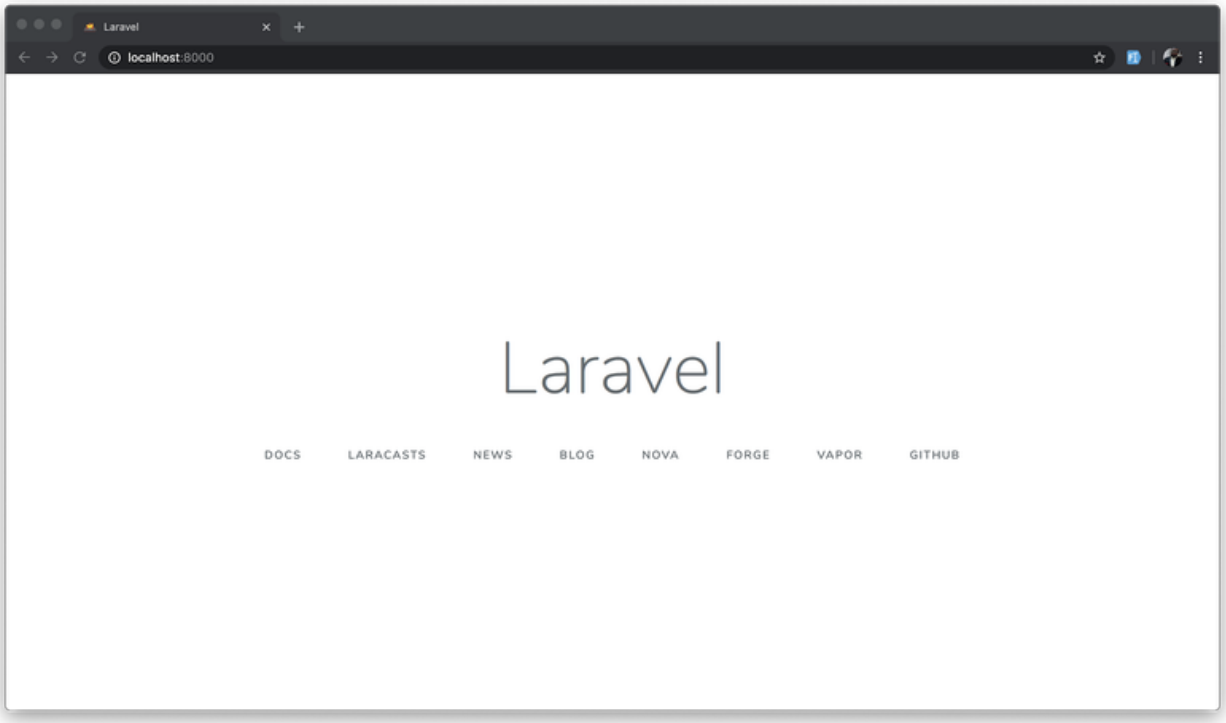
Next, create a `.env` file at the root of the project and populate it with the content found in the `.env.example` file. You can do this manually or by running the command below:

```
$ cp .env.example .env
```

Now generate the Laravel application key for this project with:

```
$ php artisan key:generate
```

You can now run the application with `php artisan serve` and proceed to http://localhost:8000 to view the homepage:



There is not much to see here as this is just a default page for a newly installed Laravel project.

Setting Up the Database

To get started with testing, you need to set up your testing database. In this tutorial, we will keep things simple by using an in-memory SQLite database. It gives the advantage of improved speed for our test scripts.

Create a `test.sqlite` file in the `database` folder. This file will be used to interact with our testing database and maintain a separate configuration from the main database. Next, replace the database environment variables from `.env.testing` in your `.env` file.

```
DB_CONNECTION=sqlite
DB_HOST=null
DB_PORT=null
DB_DATABASE=database/test.sqlite
DB_USERNAME=null
DB_PASSWORD=null
```

Each test requires migrations. We will need to run migrations before each test to properly build the database for each test. Let's configure that by opening your base `TestCase` class located

in `tests/TestCase.php` file and update it as shown below:

```

<?php

namespace Tests;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\TestCase as BaseTestCase;
use Illuminate\Support\Facades\Artisan;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication, DatabaseMigrations;

    public function setUp(): void
    {
        parent::setUp();
        Artisan::call('passport:install');
    }
}
```

Here we included the `DatabaseMigrations` trait and then added an `Artisan` call to install passport.

Lastly, use the following command to run PHPUnit from the terminal:

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

You will see the results as shown below:

```
PHPUnit 8.5.4 by Sebastian Bergmann and contributors.

..                                                                  2 ,

Time: 2.03 seconds, Memory: 24.00 MB

OK (2 tests, 2 assertions)

```

The output above showed that two tests were run successfully. These were the default tests that came installed with Laravel. We will make modifications in a bit.

To make the command for running the PHPUnit relatable, open `composer.json` file and add the test command to the scripts section as shown below:

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

Henceforth, the test command will be available as `composer test` .

Create CEO factory

Factories in Laravel make use of the Faker PHP library to conveniently generate random data for testing. Since Laravel comes preloaded with a factory definition for `User` class. We will run the following command to generate one for the `CEO` class:

```
$ php artisan make:factory CEOFactory
```

This will create a new file named `CEOFactory.php` within the `database/factories` folder. Open this new file and paste the following content in it:

```
<?php

/** @var \Illuminate\Database\Eloquent\Factory $factory */

use App\CEO;
use Faker\Generator as Faker;

$factory->define(CEO::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'company_name' => $faker->unique()->company,
        'year' => $faker->year,
        'company_headquarters' => $faker->city,
        'what_company_does' => $faker->sentence
    ];
});
```

We have specified the fields for our `CEO` table and used the Faker library to generate the correct format of random data for all the fields.

Writing our First Test

Let's start writing our test as mentioned earlier. Before that, delete the two example test files within `tests/Feature` and `tests/Unit` folders respectively.

We will begin by writing a test for the authentication process. This includes Registration and Login. We already have a controller created for that purpose within the `API` folder. So create the `AuthenticationTest` file with:

```
$ php artisan make:test AuthenticationTest
```

This will create the `AuthenticationTest.php` file inside the `test/Feature` folder. Open the new file and replace its contents with:

```php
<?php

namespace Tests\Feature;

use App\User;
use Tests\TestCase;

class AuthenticationTest extends TestCase
{
    public function testRequiredFieldsForRegistration()
    {
        $this->json('POST', 'api/register', ['Accept' => 'application/
            ->assertStatus(422)
            ->assertJson([
                "message" => "The given data was invalid.",
                "errors" => [
                    "name" => ["The name field is required."],
                    "email" => ["The email field is required."],
                    "password" => ["The password field is required."],
                ]
            ]);
    }

    public function testRepeatPassword()
    {
        $userData = [
            "name" => "John Doe",
            "email" => "doe@example.com",
            "password" => "demo12345"
        ];

        $this->json('POST', 'api/register', $userData, ['Accept' => 'a
            ->assertStatus(422)
            ->assertJson([
                "message" => "The given data was invalid.",
                "errors" => [
                    "password" => ["The password confirmation does not
                ]
            ]);
    }

    public function testSuccessfulRegistration()
```

```
    {
        $userData = [
            "name" => "John Doe",
            "email" => "doe@example.com",
            "password" => "demo12345",
            "password_confirmation" => "demo12345"
        ];

        $this->json('POST', 'api/register', $userData, ['Accept' => 'a
            ->assertStatus(201)
            ->assertJsonStructure([
                "user" => [
                    'id',
                    'name',
                    'email',
                    'created_at',
                    'updated_at',
                ],
                "access_token",
                "message"
            ]);
    }
}
```

From the file above, the following tests were written:

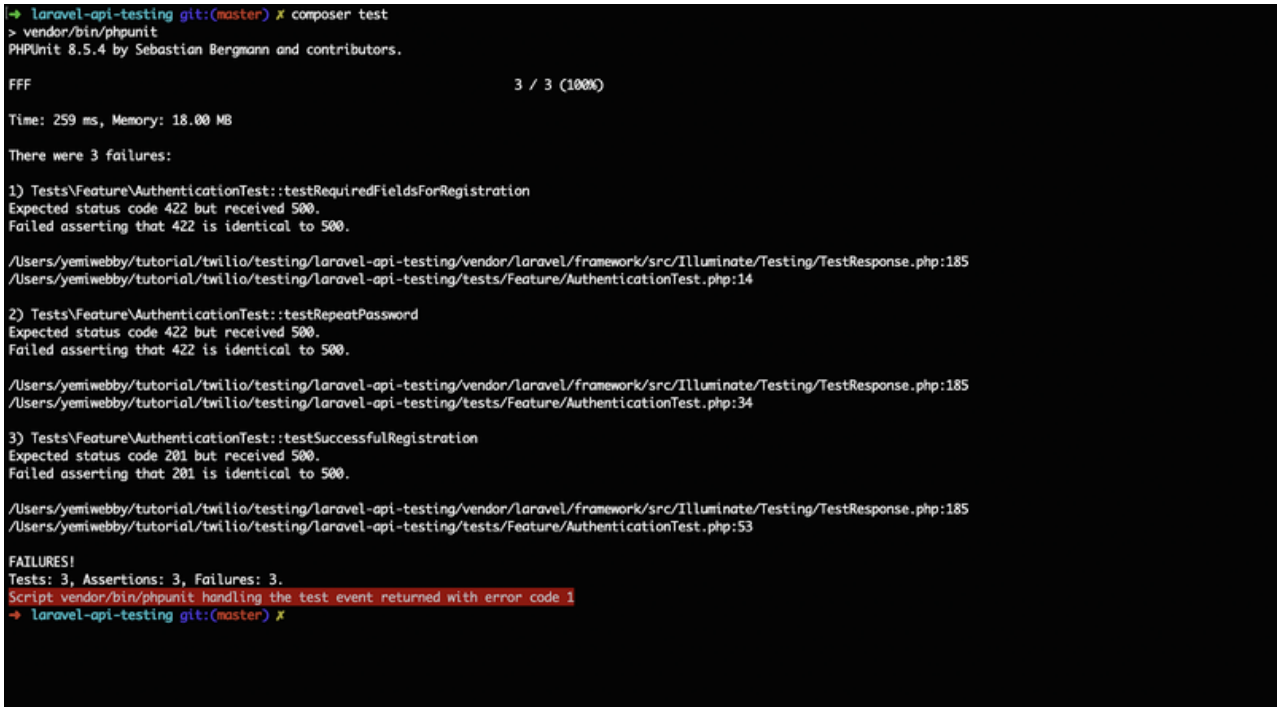
- `testRequiredFieldsForRegistration` : This test ensures that all required fields for the registration process are filled accordingly.
- `testRepeatPassword` : This mandates a user to repeat passwords. The repeated password must match the first one for this test to pass.
- `testSuccessfulRegistration` : Here, we created a user, populated with dummy data to ensure that users can sign up successfully.

Now use the following command to run our test using PHPUnit:

```
$ composer test
```

You will see results as below:

```
> vendor/bin/phpunit
PHPUnit 8.5.4 by Sebastian Bergmann and contributors.

FFF                                                                 3 /

Time: 259 ms, Memory: 18.00 MB

There were 3 failures:

1) Tests\Feature\AuthenticationTest::testRequiredFieldsForRegistration
Expected status code 422 but received 500.
Failed asserting that 422 is identical to 500.

/Users/yemiwebby/tutorial/twilio/testing/laravel-api-testing/vendor/la
/Users/yemiwebby/tutorial/twilio/testing/laravel-api-testing/tests/Fea

2) Tests\Feature\AuthenticationTest::testRepeatPassword
Expected status code 422 but received 500.
Failed asserting that 422 is identical to 500.

/Users/yemiwebby/tutorial/twilio/testing/laravel-api-testing/vendor/la
/Users/yemiwebby/tutorial/twilio/testing/laravel-api-testing/tests/Fea

3) Tests\Feature\AuthenticationTest::testSuccessfulRegistration
Expected status code 201 but received 500.
Failed asserting that 201 is identical to 500.

/Users/yemiwebby/tutorial/twilio/testing/laravel-api-testing/vendor/la
/Users/yemiwebby/tutorial/twilio/testing/laravel-api-testing/tests/Fea

FAILURES!
Tests: 3, Assertions: 3, Failures: 3.
Script vendor/bin/phpunit handling the test event returned with error c
```



This is expected as we are yet to implement the feature. Now let's write the code to make our test pass. Open `app/Http/Controllers/API/Auth/AuthController.php` and use the following content for it:

```
<?php

namespace App\Http\Controllers\API\Auth;

use App\Http\Controllers\Controller;
use App\User;
use Illuminate\Http\Request;

class AuthController extends Controller
{
    public function register(Request $request)
    {
        $validatedData = $request->validate([
            'name' => 'required|max:55',
            'email' => 'email|required|unique:users',
            'password' => 'required|confirmed'
        ]);

        $validatedData['password'] = bcrypt($request->password);

        $user = User::create($validatedData);

        $accessToken = $user->createToken('authToken')->accessToken;

        return response([ 'user' => $user, 'access_token' => $accessTol
    }
}
```

Now run `composer test`. At this point, our test should pass.

Test the Login Endpoint

Update the `AuthenticationTest.php` file by adding more methods as shown here:

```
<?php

namespace Tests\Feature;

use App\User;
use Tests\TestCase;

```

```
8  class AuthenticationTest extends TestCase
9  {
10     ...
11
12     public function testMustEnterEmailAndPassword()
13     {
14         $this->json('POST', 'api/login')
15             ->assertStatus(422)
16             ->assertJson([
17                 "message" => "The given data was invalid.",
18                 "errors" => [
19                     'email' => ["The email field is required."],
20                     'password' => ["The password field is required."],
21                 ]
22             ]);
23     }
24
25     public function testSuccessfulLogin()
26     {
27         $user = factory(User::class)->create([
28            'email' => 'sample@test.com',
29            'password' => bcrypt('sample123'),
30         ]);
31
32
33         $loginData = ['email' => 'sample@test.com', 'password' => 'sam
34
35         $this->json('POST', 'api/login', $loginData, ['Accept' => 'app
36             ->assertStatus(200)
37             ->assertJsonStructure([
38                "user" => [
39                    'id',
40                    'name',
41                    'email',
42                    'email_verified_at',
43                    'created_at',
44                    'updated_at',
45                ],
46                 "access_token",
47                 "message"
48             ]);
49
50         $this->assertAuthenticated();
51     }
52 }
```

Here we also created a test to ensure that the required fields are not left empty by the user using the `testMustEnterEmailAndPassword()` method. Within the `testSuccessfulLogin()` method, we created a dummy user to ascertain that the user is authenticated successfully.

We can now go ahead and run the test again using `composer test`. You guessed it, this will fail once more. To ensure that the test passes, update the `AuthController.php` file as follows:

```
// app/Http/Controllers/API/Auth/AuthController.php
<?php

namespace App\Http\Controllers\API\Auth;

use App\Http\Controllers\Controller;
use App\User;
use Illuminate\Http\Request;

class AuthController extends Controller
{
    ...

    public function login(Request $request)
    {
        $loginData = $request->validate([
            'email' => 'email|required',
            'password' => 'required'
        ]);

        if (!auth()->attempt($loginData)) {
            return response(['message' => 'Invalid Credentials']);
        }

        $accessToken = auth()->user()->createToken('authToken')->acces

        return response(['user' => auth()->user(), 'access_token' => $a

    }
}
```

In total, we have written five different, important tests. Some of the tested cases include the status and the `json()` structure of the response from the API. In the next section, we will

create the sets of tests for CEO endpoints.

Writing Tests for the CEO Endpoints

In this section, we will start by creating a new test file to house the test scripts for the CEO endpoints. Use the following command for that purpose:

```
$ php artisan make:test CEOTest
```

The preceding command will create a new test file named `CEOTest.php` file within the `tests/Feature` folder. Open it and replace its contents with the following:

```
<?php

namespace Tests\Feature;

use App\CEO;
use App\User;
use Tests\TestCase;

class CEOTest extends TestCase
{
    public function testCEOCreatedSuccessfully()
    {
        $user = factory(User::class)->create();
        $this->actingAs($user, 'api');

        $ceoData = [
            "name" => "Susan Wojcicki",
            "company_name" => "YouTube",
            "year" => "2014",
            "company_headquarters" => "San Bruno, California",
            "what_company_does" => "Video-sharing platform"
        ];

        $this->json('POST', 'api/ceo', $ceoData, ['Accept' => 'applica
            ->assertStatus(201)
            ->assertJson([
                "ceo" => [
                    "name" => "Susan Wojcicki",
                    "company_name" => "YouTube",
                    "year" => "2014"
```

```
                    "year" => "2014",
                    "company_headquarters" => "San Bruno, California",
                    "what_company_does" => "Video-sharing platform"
                ],
                "message" => "Created successfully"
            ]);
    }

    public function testCEOListedSuccessfully()
    {

        $user = factory(User::class)->create();
        $this->actingAs($user, 'api');

        factory(CEO::class)->create([
            "name" => "Susan Wojcicki",
            "company_name" => "YouTube",
            "year" => "2014",
            "company_headquarters" => "San Bruno, California",
            "what_company_does" => "Video-sharing platform",
        ]);

        factory(CEO::class)->create([
            "name" => "Mark Zuckerberg",
            "company_name" => "FaceBook",
            "year" => "2004",
            "company_headquarters" => "Menlo Park, California",
            "what_company_does" => "The world's largest social network'
        ]);

        $this->json('GET', 'api/ceo', ['Accept' => 'application/json'])
            ->assertStatus(200)
            ->assertJson([
                "ceos" => [
                    [
                        "id" => 1,
                        "name" => "Susan Wojcicki",
                        "company_name" => "YouTube",
                        "year" => "2014",
                        "company_headquarters" => "San Bruno, Californ:
                        "what_company_does" => "Video-sharing platform'
                    ],
                    [
                        "id" => 2,
                        "name" => "Mark Zuckerberg",
                        "company_name" => "FaceBook",
                        "year" => "2004",
```

```
                "company_headquarters" => "Menlo Park, Californ
                "what_company_does" => "The world's largest so
            ]
        ],
        "message" => "Retrieved successfully"
    ]);
    }
}
```

This may look somewhat daunting, but it is similar to the tests we wrote earlier. Let's break it down. We created two different methods:

- `testCEOCreatedSuccessfully` : To test that we can create a CEO record using the appropriate data.
- `testCEOListedSuccessfully` : Here we ensure that the list of created CEOs can be retrieved and returned as a response.

Unlike the `AuthenticationTest` , the `CEOTest` was written for endpoints that are protected by a middleware named `auth:api` . To ensure that the newly created dummy user is authenticated before accessing the endpoints, we used `$this->actingAs($user, 'api')` to authenticate and authorize such user to have access to carry out any of the CRUD (create, read, update and delete) activities.

Now run the test again using `composer test` . You will see that it fails again. By now, I am sure you understand why this wasn't successful. Just in case you're still learning the logic of these tests, we need to populate the `CEOController.php` file with the appropriate code to make the test pass.

Update the CEOController

Navigate to the `app/Http/Controllers/API/CEOController.php` file and use the following content for it:

```
<?php
```

```php
namespace App\Http\Controllers\API;

use App\CEO;
use App\Http\Controllers\Controller;
use App\Http\Resources\CEOResource;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class CEOController extends Controller
{
    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $data = $request->all();

        $validator = Validator::make($data, [
            'name' => 'required|max:255',
            'company_name' => 'required|max:255',
            'year' => 'required|max:255',
            'company_headquarters' => 'required|max:255',
            'what_company_does' => 'required'
        ]);

        if($validator->fails()){
            return response(['error' => $validator->errors(), 'Validat:
        }

        $ceo = CEO::create($data);

        return response([ 'ceo' => new CEOResource($ceo), 'message' =>
    }


    public function index()
    {
        $ceos = CEO::all();

        return response([ 'ceos' => CEOResource::collection($ceos), 'm
    }
}
```

Here, we created methods to store the records of a new *CEO* and also retrieve the list of `CEOs` from the database respectively. You can run the test again and discover it passes this time.

Lastly, let's add more tests to retrieve, update, and also delete the details of a particular CEO. Open the `CEOTest.php` file and add the following methods:

```
<?php

namespace Tests\Feature;

use App\CEO;
use App\User;
use Tests\TestCase;

class CEOTest extends TestCase
{
    ...

    public function testRetrieveCEOSuccessfully()
    {
        $user = factory(User::class)->create();
        $this->actingAs($user, 'api');

        $ceo = factory(CEO::class)->create([
            "name" => "Susan Wojcicki",
            "company_name" => "YouTube",
            "year" => "2014",
            "company_headquarters" => "San Bruno, California",
            "what_company_does" => "Video-sharing platform"
        ]);

        $this->json('GET', 'api/ceo/' . $ceo->id, [], ['Accept' => 'apl
            ->assertStatus(200)
            ->assertJson([
                "ceo" => [
                    "name" => "Susan Wojcicki",
                    "company_name" => "YouTube",
                    "year" => "2014",
                    "company_headquarters" => "San Bruno, California",
                    "what_company_does" => "Video-sharing platform"
                ],
                "message" => "Retrieved successfully"
            ]);
    }
```

```
    }

    public function testCEOUpdatedSuccessfully()
    {
        $user = factory(User::class)->create();
        $this->actingAs($user, 'api');

        $ceo = factory(CEO::class)->create([
            "name" => "Susan Wojcicki",
            "company_name" => "YouTube",
            "year" => "2014",
            "company_headquarters" => "San Bruno, California",
            "what_company_does" => "Video-sharing platform"
        ]);

        $payload = [
            "name" => "Demo User",
            "company_name" => "Sample Company",
            "year" => "2014",
            "company_headquarters" => "San Bruno, California",
            "what_company_does" => "Video-sharing platform"
        ];

        $this->json('PATCH', 'api/ceo/' . $ceo->id , $payload, ['Accep
            ->assertStatus(200)
            ->assertJson([
                "ceo" => [
                    "name" => "Demo User",
                    "company_name" => "Sample Company",
                    "year" => "2014",
                    "company_headquarters" => "San Bruno, California",
                    "what_company_does" => "Video-sharing platform"
                ],
                "message" => "Updated successfully"
            ]);
    }

    public function testDeleteCEO()
    {
        $user = factory(User::class)->create();
        $this->actingAs($user, 'api');

        $ceo = factory(CEO::class)->create([
            "name" => "Susan Wojcicki",
            "company_name" => "YouTube",
            "year" => "2014",
            "company_headquarters" => "San Bruno, California",
```

```
            "what_company_does" => "Video-sharing platform"
        ]);

        $this->json('DELETE', 'api/ceo/' . $ceo->id, [], ['Accept' =>
            ->assertStatus(204);
    }

}
```

The above written tests were used to target the records of a particular `CEO` by passing a unique `id` as a parameter to the `api/ceo` endpoints using the appropriate HTTP verbs (i.e `GET`, `PATCH`, `DELETE`).

Next, we will update the `CEOController` file to ensure that the new tests pass as well. Open the `app/Http/Controllers/API/CEOController.php` and include the following methods:

```
<?php

namespace App\Http\Controllers\API;

use App\CEO;
use App\Http\Controllers\Controller;
use App\Http\Resources\CEOResource;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class CEOController extends Controller
{
    ...

    /**
     * Display the specified resource.
     *
     * @param  \App\CEO  $ceo
     * @return \Illuminate\Http\Response
     */
    public function show(CEO $ceo)
    {
        return response([ 'ceo' => new CEOResource($ceo), 'message' =>

    }

    /**
```

```
28          * Update the specified resource in storage.
29          *
30          * @param  \Illuminate\Http\Request  $request
31          * @param  \App\CEO  $ceo
32          * @return \Illuminate\Http\Response
33          */
34         public function update(Request $request, CEO $ceo)
35         {
36
37             $ceo->update($request->all());
38
39             return response([ 'ceo' => new CEOResource($ceo), 'message' =>
40         }
41
42         /**
43          * Remove the specified resource from storage.
44          *
45          * @param \App\CEO $ceo
46          * @return \Illuminate\Http\Response
47          * @throws \Exception
48          */
49
50
51         public function destroy(CEO $ceo)
52         {
53             $ceo->delete();
54
55             return response(['message' => 'Deleted'], 204);
56         }
57     }
```

The methods created here will retrieve, update, and delete the records of a CEO from the database.

We can now run the test command for the last time using `composer test` and you will see the following output:

```
> vendor/bin/phpunit
PHPUnit 8.5.4 by Sebastian Bergmann and contributors.

..........                                                        10 /

Time: 915 ms, Memory: 32.00 MB

OK (10 tests, 35 assertions)
```

Conclusion

In this tutorial, we were able to use a test-driven development approach to write a couple of tests for the endpoints in our Laravel API project. You will further appreciate test-driven development when you discover that once you keep adding more features, the new implementations do not in any way interrupt the functioning of your existing codebase.

The complete codebase for the tutorial can be found here on GitHub. You can explore, add more tests, and write the code that will enable your test to pass accordingly.

Olususi Oluyemi is a tech enthusiast, programming freak, and a web development junkie who loves to embrace new technology.

- Twitter: https://twitter.com/yemiwebby
- GitHub: https://github.com/yemiwebby
- Website: https://yemiwebby.com.ng/

RATE THIS POST ★★★★★

AUTHORS |  Oluyemi Olususi

REVIEWERS | Marcus Battle

Search

POSTS BY STACK

JAVA .NET RUBY PHP PYTHON SWIFT ARDUINO JAVASCRIPT

POSTS BY PRODUCT

SMS AUTHY VOICE TWILIO CLIENT MMS VIDEO TASK ROUTER FLEX SIP IOT PROGRAMMABLE CHAT STUDIO

CATEGORIES

Code, Tutorials and Hacks

Customer Highlights

Developers Drawing The Owl

News

Stories From The Road

The Owl's Nest: Inside Twilio

Get started