

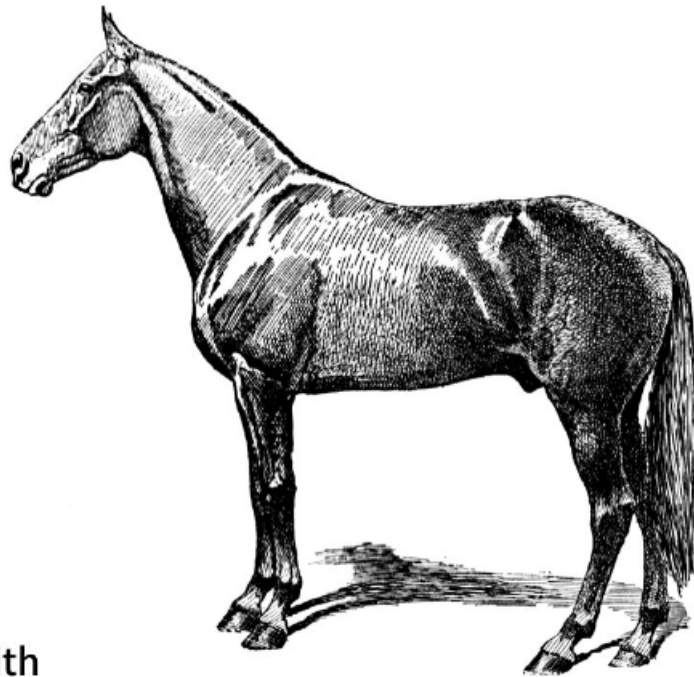
Laravel and APIs

Software as a Service - Back-End Development

Session 01

Developed by Adrian Gould

There's an API for that



Coping With

**SAAS
Addiction**

O'RLY?

H.E.Addicted

Contents

- [Laravel and APIs](#)
- [Requirements](#)
- [Creating a New Laravel API Application](#)
 - [Laravel and .env, APPENV & APPDEBUG](#)
 - [Package Installation](#)
 - [Laravel Pint](#)
 - [Laravel Sanctum](#)
 - [Spatie Pest Test Time](#)
 - [Scribe Documentation Generator](#)
- [Enabling API Routes](#)
- [Building the Region Feature](#)
 - [Region Model](#)
 - [Region Migration](#)
 - [Region Seeder](#)
- [Factories](#)
 - [Create the Region Factory](#)
 - [Migrate and Seed](#)
- [Creating our first API Endpoint](#)
 - [General API Response Class](#)
 - [Constructor](#)
 - [Rollback](#)
 - [Throw](#)
 - [Send Response](#)
 - [Region API Controller](#)
 - [Create the Controller](#)
 - [Edit the Controller Class](#)
 - [Create the Routes](#)
 - [Listing the Routes](#)
- [Testing our Endpoint](#)
 - [Red and Green](#)
 - [Manual Testing Route via Browser](#)
 - [Unit Tests vs Feature Tests](#)
 - [Unit test](#)
 - [Feature tests](#)
 - [Testing with PEST](#)
 - [Example Test](#)
 - [Writing the Region Controller index \(Browse\) Test](#)
- [Documenting an Endpoint](#)
 - [Configuring Scribe](#)
 - [Adding Documentation to the API](#)
 - [Generating and Publishing the API Docs](#)
 - [Autogenerating Documentation](#)
- [Creating another endpoint](#)
 - [1\) Create New Test](#)
 - [2\) Execute the Tests](#)
 - [Executing from the command line](#)
 - [Executing Tests in PhpStorm](#)

- [Alternative Ways To Run Tests in PhpStorm](#)
 - [3\) Write Code](#)
 - [4\) Rerun Tests](#)
 - [5\) Fix the error and go back to step 4](#)
 - [6\) Refactor Code](#)
 - [7\) Document](#)
 - [What about the Route Refactoring?](#)
 - [What about Automatic Test Execution?](#)
 - [PhpStorm Test Runner Icons](#)
 - [Exercises](#)
 - [Exercise 1: Subregion](#)
 - [Subregion model details](#)
 - [Add Documentation](#)
 - [Exercise 2: Country](#)
 - [Country model details](#)
 - [Add Documentation](#)
 - [Exercise 3: States](#)
 - [State model details](#)
 - [Add Documentation](#)
 - [Exercise 4: Cities](#)
 - [City model details](#)
 - [Add Documentation](#)
 - [Questions to Investigate](#)
 - [END](#)
-

Requirements

Please see the Development Environments article for details on what we use for developing and teaching at NM TAFE.

Creating a New Laravel API Application

Open Windows Terminal (or equivalent)

Update the laravel installer and any other global packages:

```
composer global upgrade --no-interaction
```

Create a new Laravel application:

```
laravel new
```

When asked, answer these questions:

- What is the name of your project? **workopia-api-xxx**
xxx is replaced by your initials.
- Would you like to install a starter kit? **No Starter Kit**
- Which testing framework do you prefer? **Pest**
- Would you like to initialise a Git repository? **Yes**
- Which database will your application use? **SQLite** (you may opt to use another DBMS if you wish)
- Would you like to run the default database migrations? **Yes**
- The SQLite database ... Would you like to create it? **Yes**

Move into the new application folder:

```
cd workopia-api-xxx
```

Laravel and `.env` , `APP_ENV` & `APP_DEBUG`

In your `.env` file you will note that there is a variable called `APP_ENV` . This is the application's execution environment. There are two main `APP_ENV` values:

- `local`
- `production`

During **development** we ALWAYS use `local` and when the application is **live** we use `production` .

To go along with this there is the `APP_DEBUG` variable, which turns debugging mode on and off.

In practice, `APP_DEBUG` is set to:

- `true` when `APP_ENV` is `local`
- `false` when `APP_ENV` is `production`

You could use a configuration that allows for "testing" in a sandbox with a copy of production data:

```
APP_ENV=production
APP_DEBUG=true
```

You will often keep a local and production `.env` file so as to make sure you enable the correct environment.

When learning, you could copy the `.env` to `.env.local` and `.env.production` . You could even have one for staging/testing, `.env.staging` and `.env.testing` .

Remember that in reality you would **NOT** version control a `.env` file as it is a security risk.

Package Installation

Before we go any further we will add some packages ready for later development, including:

- Laravel Sanctum - for authentication of users.
- Scribe - an API documentation generator.
- Laravel Pint (if not already installed) - a code linter.

Laravel Pint

Install Pint using the following steps:

```
composer require --dev laravel/pint
```

It is then possible to integrate Pint into PhpStorm so that on commit your code is tidied up.

Laravel Sanctum

Add Laravel's Sanctum authentication package:

```
composer require laravel/sanctum
```

Spatie Pest Test Time

Spatie's Pest test time allows you to 'freeze time' when testing. this can be exceptionally useful, and you will discover.

```
composer require spatie/pest-plugin-test-time --dev
```

Scribe Documentation Generator

Install using composer then publish the configuration file.

```
composer require --dev knuckleswtf/scribe  
php artisan vendor:publish --tag=scribe-config
```

We will leave configuration for a later stage.

Enabling API Routes

First enable the API routing and other required functionality:

```
php artisan install:api
```

When it asks, say yes to running any new migrations.

Building the Region Feature

To show how we go about the basics of creating a feature, we will look at building a region lookup which uses the "continents" contained on the globe.

We will create the index and show API routes only for this example.

We will also show how we can use JSON data to provide our seed data.

To make it easy on ourselves we will generate all of the parts we will need in one command. In each section, though, we have given the individual command for generating the required item, if you are completing this piece by piece.

```
php artisan make:model Region --all --api
```

Region Model

Let's get ourselves into the development, and create and edit the Region Model

```
php artisan make:model Region
```

Remember: We created all the components earlier, the command is there for reference only.

Edit the Region model, and add the following:

```
class Region extends Model
{
    use HasFactory;

    /**
     * The attributes that are mass assignable.
     *
     * @var array<int, string>
     */
    protected $fillable = [
        'name',
    ];

    /**
     * The attributes that should be hidden for serialization.
     *
     * @var array<int, string>
     */
    protected $hidden = [
    ];
}
```

```

/**
 * Get the attributes that should be cast.
 *
 * @return array<string, string>
 */
protected function casts(): array
{
    return [

    ];
}

// TODO: Region has countries
// TODO: Region has translations (polymorphic?)
}

```

Region Migration

So we know we need one field: `name`. Possibly the simplest of all models to have!

Create the migration:

```
php artisan make:migration create_regions_table
```

Edit the new migration file:

```

// TODO: How to handle translations of the names? Possible translations table?

Schema::create('regions', function (Blueprint $table) {
    $table->id();
    $table->string('name')->index();
    $table->timestamps();
});

```

Region Seeder

Now we can create the seeder for the Model.

We are going to be a little fancy here and import the data from a JSON file that is stored in a folder `/database/data/`. If you do not have the folder, then create it.

Then download the data file from <https://github.com/dr5hn/countries-states-cities-database/blob/master/regions.json> and move it into the `database/data` folder.

Now we are ready to create the seeder.

```
php artisan make:seeder RegionSeeder
```

Edit the file and add the following code:

```

/**
 * Run the database seeds.
 *
 * Code inspired by:
 * https://www.itsolutionstuff.com/post/how-to-create-seeder-with-json-data-in-

```



```

laravelexample.html
* Data from:
* https://github.com/dr5hn/countries-states-cities-database
*/
public function run(): void
{
    Region::truncate();

    $json = File::get('database/data/regions.json');

    $countries = json_decode($json);

    foreach ($countries as $key => $value) {
        // TODO: this will need to be extended to provide for translation import.
        Region::create([
            'name' => $value->name,
        ]);
    }
}

```

Future Enhancement: As an administrator I would like to automate the update of the regions by downloading the data directly from the above web link and saving to application storage, then for the data to be imported automatically.

Factories

Factories are a great way to create fake data when testing. This is going to be very important as we develop our quick API for demonstration purposes.

In future parts we will use factories extensively to fake our data and speed up testing.

Create the Region Factory

Create a new region factory using:

```
php artisan make:factory RegionFactory
```

Open the factory file and edit/add the following:

```

<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use App\Models\Region;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Model>
 */
class RegionFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            'name' => $this->faker->name(),
        ];
    }
}

```

```
{  
    return [  
        'name' => $this->faker->words(3),  
    ];  
}
```

What we have done here is provide a way to add dummy data into fields that are missing.

This is great for testing.

Migrate and Seed

Now it is time to migrate and seed the database.

```
php artisan migrate:fresh --seed
```

Remember: The fresh in this command destroys all the database tables and recreates them, so it should only be used in development!

We are now ready to start developing the feature.

Creating our first API Endpoint

As a way of learning how to write an endpoint, we are going to create our own API endpoints for the World package.

We will create API endpoints for:

- `/api/v1/regions` - to retrieve all regions
- `/api/v1/regions/{id}` - to retrieve the region with `id`

General API Response Class

We are going to create a generalised API response class so that we do not have to manipulate the output each time we need to send a response back to the client. This is DRYing our code.

```
php artisan make:class /Classes/ApiResponseClass
```

Edit this class and add the following code (we show it in stages, and describe what it is used for).

Constructor

First is the constructor (`__construct`) method for the `ApiResponseClass` . This is automatically added by the make command.

```
class ApiResponseClass
{
    /**
     * Create a new class instance.
     * TODO: Delete this method
     */
    public function __construct()
    {
        //
    }
}
```

It's empty, so delete this method as it is not needed. We will not be instantiating any objects using this class.

Rollback

The `rollback` method is defined to allow for undoing a transaction. It is best practice to use database transactions when dealing with data insertion, updates or deletes.

When the rollback is called it will throw an error and message.

We declare the method as being static as we do not need to instantiate the class to use these methods.

```
/**
 * TODO: Add description and parameters for this method
 */
public static function rollback($e, $message = "Something went wrong! Process not
completed")
{
    DB::rollBack();
    self::throw($e, $message);
}
```

Throw

The `throw` method is used by the class to log the error to file (or elsewhere), and then to send a response to the client in JSON format, with the error code 500 and the message.

We declare the method as being static as we do not need to instantiate the class to use these methods.

```
/**
 * TODO: Add description and parameters for this method
 */
public static function throw($e, $message = "Something went wrong! Process not
completed")
{
    Log::info($e);
    throw new HttpResponseException(response()->json(["message" => $message], 500));
}
```

Send Response

Finally we get to where we actually send the response, the `sendResponse` method.

We pass the data to be sent to the client, a success message, and the required result code (default 200, OK) to the method, and it constructs an array with these items.

The response is then sent back to the client in JSON format.

We declare the method as being static as we do not need to instantiate the class to use these methods.

```
/**
 * TODO: Add description and parameters for this method
 */
public static function sendResponse($result, $message, $code = 200)
{
    $response = [
        'success' => true,
        'message'=>$message??null,
        'data' => $result
    ];
    return response()->json($response, $code);
}

/**
 * End of ApiResponseClass
 */
}
```

This generalised response class can be reused over and over again - a very useful tool in your arsenal.

You will see how it is used within the API Controller.

Region API Controller

Now we are ready to create our first Controller that will be part of the API.

There are many ways to go about this, but if we separate the API from a management application, then we can use the same naming as previously done. That is RegionController, UserController, etc.

Some people would advocate the use of an API namespace for the ability to have a single code base. This could become a very large monolithic application by doing this.

We are splitting the code base for our learning, and concentrating on the API.

Create the Controller

Unlike creating a standard resourceful controller we will tell artisan to make an API controller using the `--api` switch rather than the `--resource` switch.

Create the Region API controller:

```
php artisan make:controller RegionController --api
```

Edit the Controller Class

Next edit the `RegionController` class, by updating the `index` method.

We are going to retrieve all the regions. We are not going to filter, or paginate the results.

```
public function index()
{
    $regions = Region::all();
    return ApiResponseClass::sendResponse($regions, "regions retrieved successfully.",
    200);
}
```

Note how we use our `ApiResponseClass` to send the response.

The regions are a collection (array) of results, and the message is set so that we could take this and report to the user if needed.

The response code is `200 OK` - but this could be omitted as the default for the `sendResponse` method is `200`.

Create the Routes

We are ready to now hook the controller and the routing up so we can make use of it.

Open the `api.php` file in the `routes` folder, which was created when we told artisan to `install:api`.

Create a new route for us to use the RegionController API index method:

```
Route::group(['prefix' => 'v1'], function () {
    Route::get('regions', [RegionController::class, 'index'])->name('region.index');
    Route::get('regions/{id}', [RegionController::class, 'show'])->name('region.show');
});
```

What's special about this?

- We group these routes into `v1` of our API.

This means we will visit an endpoint that looks similar to these:

```
https://somedomain.com/api/v1/regions
```

```
https://somedomain.com/api/v1/regions/45
```

What if we are doing more than just an `index` and `show`... in that situation we can use the `Api Resource` method from the `Route` class to automatically add `index`, `show`, `create`, `update` and `delete` methods to the routes. This is very similar to using the `Route Resource` from the front end side.

Even with just two routes, we will refactor this later as we are able to make our code more readable (after adding the `show` route.)

Listing the Routes

Use the `php artisan route:list` command to see the routes you now have.

You should find the route listed, plus others generated by other components we installed previously.

VERB	Endpoint	Alias	Method
GET\ HEAD	api/v1/regions	api.regions.index	RegionController@index
GET\ HEAD	api/v1/regions/{id}	api.regions.show	RegionController@show

Testing our Endpoint

We can perform testing in a number of ways:

- Manual testing using a browser
- Manual testing using the CLI and the `curl` command
- Unit testing using Pest or similar
- Testing via a 3rd party GUI (eg. Postman)

Each has pros and cons, for example:

- Manual testing using a browser
 - may not be able to test when it comes to authenticated users
- Manual testing using the CLI and the `curl` command
 - ...
- Unit testing using Pest or similar
 - ...
- Testing via a 3rd party GUI (eg. Postman)
 - ...

Red and Green

Unit testing has two states:

- Red
 - Test errors.
- Green
 - Test passes.

You are aiming for ALL GREEN!

:LiAlertTriangle: **Note:** Images will show details from both Region and Country. We will add Country later.

```

~/Source/Repos/workopia-api-xxx git:(main)±58 (1.46s)
php artisan test --profile

PASS Tests\Unit\CountryControllerTest
✓ countrycontroller 0.48s
✓ it can fetch all countries 0.05s
✓ it can fetch one country by ID 0.01s
✓ it can fetch one country by ISO-2 code 0.01s

PASS Tests\Unit\ExampleTest
✓ that true is true

PASS Tests\Feature\ExampleTest
✓ it has welcome page 0.03s

Tests: 6 passed (9 assertions)
Duration: 0.73s

Top 10 slowest tests:
Tests\Unit\CountryControllerTest > countrycontroller 0.48s
Tests\Unit\CountryControllerTest > it can fetch all countries 0.05s
Tests\Feature\ExampleTest > it has welcome page 0.03s
Tests\Unit\CountryControllerTest > it can fetch one country by ISO-2 code 0.01s
Tests\Unit\CountryControllerTest > it can fetch one country by ID 0.01s
Tests\Unit\ExampleTest > that true is true 0.00s

(80.87% of 0.73s) 0.59s
  
```

Above is an image from running tests on the command line, and giving an ALL GREEN response.

Manual Testing Route via Browser

Ok, let's try our new index route.

Open a browsers and go to: `http://workopia-api-xxx.test/api/v1/regions` and you should get data similar to (this is an extract as there are 6 regions in the database):

```
{
  "success": true,
  "message": "regions retrieved successfully.",
  "data": [
    {
      "id": 1,
      "name": "Africa",
      "created_at": "2024-07-08T14:25:50.000000Z",
      "updated_at": "2024-07-08T14:25:50.000000Z"
    },
    {
      "id": 2,
      "name": "Americas",
      "created_at": "2024-07-08T14:25:50.000000Z",
      "updated_at": "2024-07-08T14:25:50.000000Z"
    },
    {
      "id": 3,
      "name": "Asia",
      "created_at": "2024-07-08T14:25:50.000000Z",
      "updated_at": "2024-07-08T14:25:50.000000Z"
    },
    {
      "id": 4,
      "name": "Europe",
      "created_at": "2024-07-08T14:25:50.000000Z",
      "updated_at": "2024-07-08T14:25:50.000000Z"
    },
    {
      "id": 5,
      "name": "Oceania",
      "created_at": "2024-07-08T14:25:50.000000Z",
      "updated_at": "2024-07-08T14:25:50.000000Z"
    },
    {
      "id": 6,
      "name": "Polar",
      "created_at": "2024-07-08T14:25:50.000000Z",
      "updated_at": "2024-07-08T14:25:50.000000Z"
    }
  ]
}
```

Note: You will probably find your display will not be split into lines as we have above.

Unit Tests vs Feature Tests

There are multiple types of tests that you can write, with the concept of writing tests being generally seen as "unit testing". With testing there are many different types of tests, and they include:

1. Unit tests
2. Feature tests

as well as other types of test including:

3. User testing
4. Regression testing
5. and so on...

Unit test

A unit test is responsible for testing a part of your application in isolation, meaning not how it interacts with everything else, but how it processes data and how it handles various scenarios, you can look at it as how a class for example is supposed to work from a developer's point of view.

Feature tests

Feature tests are black box tests, and they test a part of an application from end to end, for example an API, or a group of actions etc.

Feature tests are more from the users perspective rather than the developer's, let's say we run blog, our blog uses an SPA, so from the backend we just need an API, and we want to create an endpoint to create replies to comments, let's also assume that we already have this implementation of the `Comment` model.

Feature and Unit tests, what they are and how to use them in Laravel. (2022). Abedt.com.

<https://abedt.com/blog/feature-tests-vs-unit-tests/>

Testing with PEST

Pest is a unit testing framework that is built upon the PHPUnit framework. It has a very expressive syntax.

:LiAlertTriangle: Note:

Whilst we are developing and using Pest based testing, we will be 'refreshing' the database regularly. This means that any existing data will be destroyed before tests are executed.

You **must not** use test systems on a `production` application.

Testing **must always be** completed on the `local` development or in the CI/CD testing phase.

To begin we will look at the example test, and then build a test for the above method.

In practice, though, we write the tests FIRST.

Example Test

Open the `tests/Feature/ExampleTest.php` file.

In here you will see:

```
<?php

it('returns a successful response', function () {
    $response = $this->get('/');

    $response->assertStatus(200);
});
```

This is a Pest test, but it is longer than it needs to be, so replace the content of the file with:

```
<?php

it('has welcome page')->get('/')->assertStatus(200);
```

Do you think that syntax reads really nicely?

Writing the Region Controller `index` (Browse) Test

Create the RegionControllerTest using:

```
php artisan pest:test BrowseRegionsTest --feature
```

Opening this file reveals the following:

```
<?php

test('Regioncontroller', function () {
    expect(true)->toBeTrue();
});
```

Let's make some changes here.

First we need to add a `use` line to automatically refresh the database (yes it does a `migrate:fresh` between each test!). After that, we add a `uses` line to include the test case and refresh database classes.

```
use Illuminate\Foundation\Testing\RefreshDatabase;
use function Spatie\PestPluginTestTime\testTime;

uses(Tests\TestCase::class, RefreshDatabase::class);

// Freeze time!
testTime()->freeze('2024-07-08 14:25:50');
```

After the above test (we will leave it there for now), add the following test code:

```
it('can fetch all regions', function () {

    // Test data
    $data = [
        'success' => true,
        'message' => 'regions retrieved successfully.',
        'data' => [
            [
                'id' => 1,
                'name' => 'World Region 1',
                'created_at' => '2024-07-08T14:25:50.000000Z',
                'updated_at' => '2024-07-08T14:25:50.000000Z',
            ], [
                'id' => 2,
                'name' => 'World Region 2',
                'created_at' => '2024-07-08T14:25:50.000000Z',
                'updated_at' => '2024-07-08T14:25:50.000000Z',
            ],
        ],
    ];

    // Insert the above test data into the Region model
    foreach ($data['data'] as $datum) {
        Region::create($datum);
    }

    // Get a response from calling the API endpoint
    $response = $this->getJson('/api/v1/regions');
    // Check the response was as expected:
```

```
// - 200 OK
// - The correct JSON data (matches the sample data)
$response->assertStatus(200)->assertJson($data);
});
```

Documenting an Endpoint

As you saw in the set-up of the application we installed the Knuckle SWTF Scribe package. This package provides us with a robust way to document our API.

Just in case you forgot to install the Scribe package execute the following:

```
composer require --dev knuckleswtf/scribe
php artisan vendor:publish --tag=scribe-config
```

Configuring Scribe

Open the configuration file in PhpStorm (`shift shift` type in `apidoc-` and find the `config/scribe.php` file.)

Make these changes, with strings being in quotes, and booleans being unquoted...

- `type: laravel,`
- `title: Workopia API,`
- `description: Developed by YOUR_NAME_HERE,`
- `use_csrf: true,`
- `example_languages: bash, javascript, php, python`

This configuration means that Scribe will use the Laravel routing and ...

It will also add CSRF to form requests, and automatically generate examples of the API using Bash (curl), JavaScript, PHP and Python.

Adding Documentation to the API

Adding documentation to the API is done by using PHP Doc block comments.

Here is an example:

```
/**
 * Return all regions
 *
 * @group Region
 *
 * @response status=200 scenario="Region Found" {
 *   "success": true,
 *   "message": "regions retrieved successfully.",
 *   "data": [
 *     {
 *       'id' => 1,
 *       'name' => 'World Region 1',
 *       'created_at' => '2024-07-08T14:25:50.000000Z',
 *       'updated_at' => '2024-07-08T14:25:50.000000Z',
 *     }, {
 *       'id' => 2,
 *       'name' => 'World Region 2',
 *       'created_at' => '2024-07-08T14:25:50.000000Z',
 *       'updated_at' => '2024-07-08T14:25:50.000000Z',
 *     },
 *   ]
 * }
 *
 * @return \Illuminate\Http\JsonResponse
 */
public function index()
{
    $regions = Region::all();
    return ApiResponseClass::sendResponse($regions, "regions retrieved successfully.",
200);
}
```

You are probably thinking, OMG!

Yes, in the example above the documentation via the PHP Doc comments is 5x the code for the method itself.

The reason for this is that the comments contain:

- Grouping - to keep related API documentation together and make it easier to locate.
- Response - is an example response for display in the documentation without accessing live data.

In the above example, there is no need to add an parameters as the method does not contain them.

:LiAlertTriangle: Note that you DO NOT remove the method documentation such as the `@return` comment as this is used by the IDE to assist you when building the application.

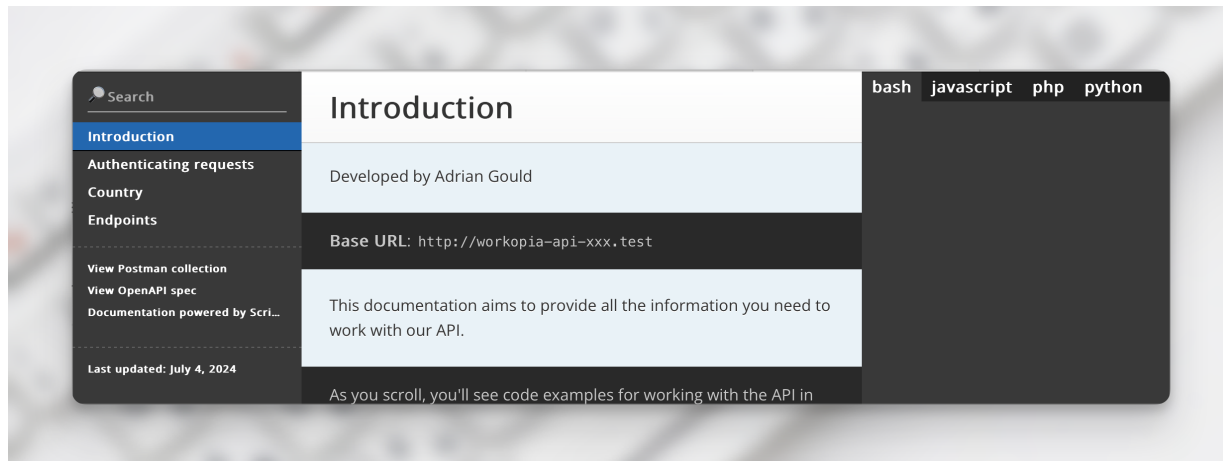
Generating and Publishing the API Docs

To generate the Docs you will perform an artisan command:

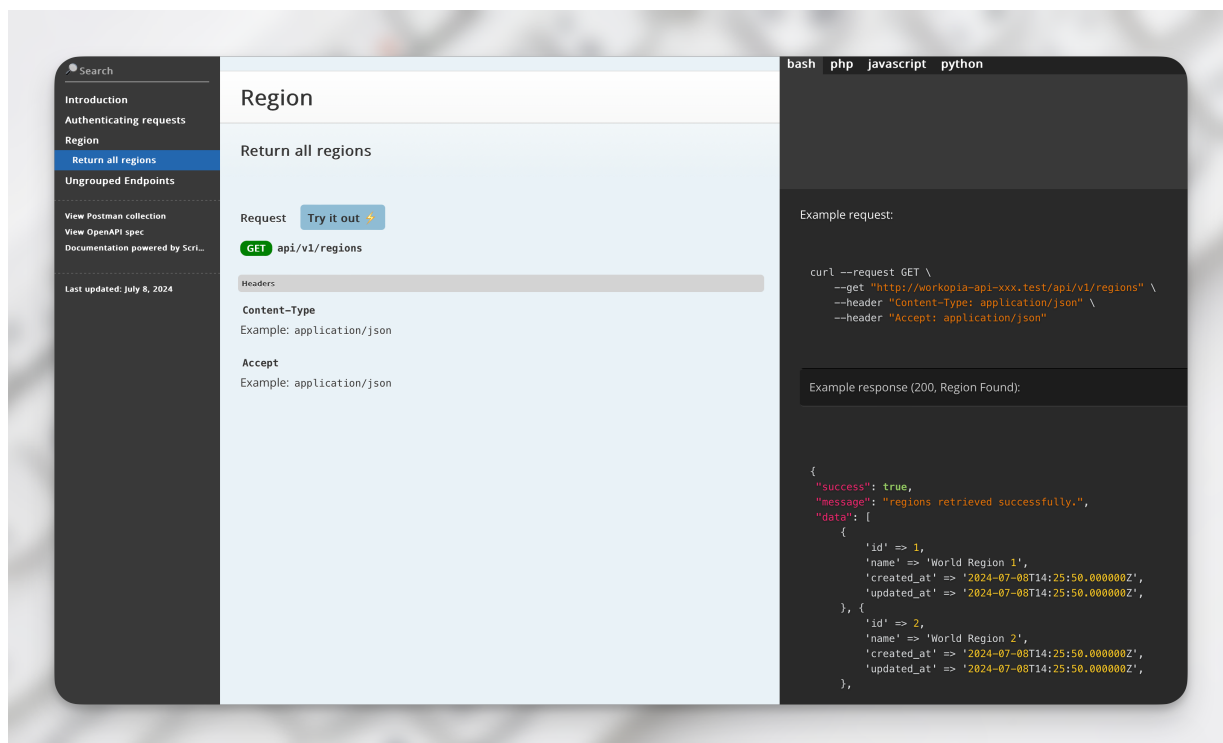
```
php artisan scribe:generate
```

You should now be able to view the documentation at: <http://workopia-api-xxx.test/docs>

Example of API Documentation Home Page



Example of Grouped API Endpoints



Autogenerating Documentation

When using Scribe, you could have it automatically regenerate the docs by using GitHub's Workflows, otherwise you will need to generate the documentation using the command we used previously.

This time we will reverse the process, and create the test FIRST, execute to see it fail, then add code to make it pass.

1) Create New Test

- What should the test be named? `GetRegionTest`
- Which type of test would you like? `Feature`

```
INFO [tests/Feature/GetRegionTest.php] created successfully.
```

```
use App\Models\Region;
use Illuminate\Foundation\Testing\RefreshDatabase;
use function Spatie\PestPluginTestTime\testTime;

uses(RefreshDatabase::class);

testTime()->freeze('2024-07-08 14:25:50');

it('can fetch one Region by ID', function () {
    $region1 =

    $region2=
        $data = [
            'success' => true,
            'message' => 'regions retrieved successfully.',
            'data' => [
                [
                    'id' => 1,
                    'name' => 'World Region 1',
                    'created_at' => '2024-07-08T14:25:50.000000Z',
                    'updated_at' => '2024-07-08T14:25:50.000000Z',
                ]
            ]
        ]
    );
});
```

```

        ], [
            'id' => 2,
            'name' => 'World Region 2',
            'created_at' => '2024-07-08T14:25:50.000000Z',
            'updated_at' => '2024-07-08T14:25:50.000000Z',
        ],
    ],
];

// Add test data
foreach ($data['data'] as $datum) {
    Region::create($datum);
}

$response = $this->getJson('/api/v1/regions/2');
$response->assertStatus(200)->assertJson($data);
});

```

We are purposely adding two items of test data so we can check to see if it returns the correct results.

2) Execute the Tests

You may execute the tests in two ways:

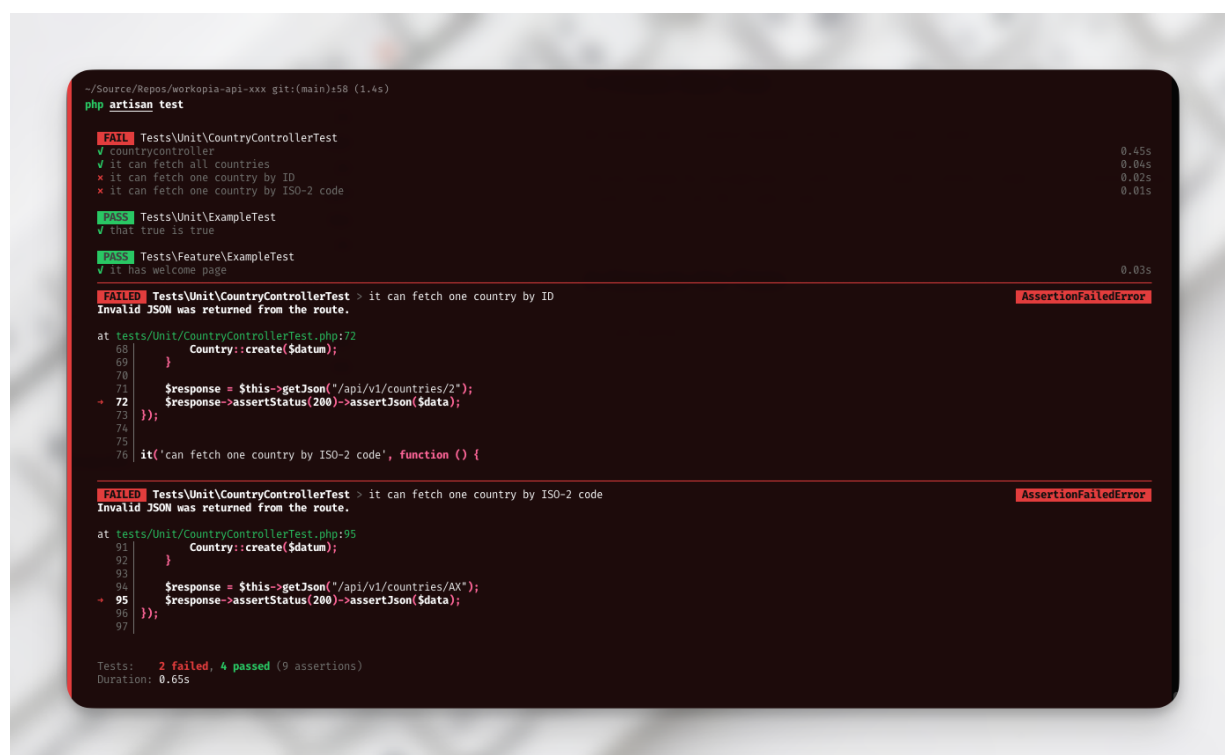
1. Command Line
2. Within PhpStorm

Executing from the command line

To execute the tests from the command line use:

```
php artisan test
```

The result will be information about the tests that passed and those that failed (screenshot for illustration only):



```
~/Source/Repos/workopia-api-xxx git:(main)58 (1.4s)
php artisan test

FAIL Tests\Unit\CountryControllerTest
✓ countrycontroller 0.45s
✓ it can fetch all countries 0.04s
✗ it can fetch one country by ID 0.02s
✗ it can fetch one country by ISO-2 code 0.01s

PASS Tests\Unit\ExampleTest
✓ that true is true

PASS Tests\Feature\ExampleTest
✓ it has welcome page 0.03s

FAILED Tests\Unit\CountryControllerTest > it can fetch one country by ID AssertionFailedError
Invalid JSON was returned from the route.

at tests/Unit/CountryControllerTest.php:72
68 Country::create($datum);
69 }
70
71 $response = $this->getJson("/api/v1/countries/2");
→ 72 $response->assertStatus(200)->assertJson($data);
73 });
74
75
76 it('can fetch one country by ISO-2 code', function () {

FAILED Tests\Unit\CountryControllerTest > it can fetch one country by ISO-2 code AssertionFailedError
Invalid JSON was returned from the route.

at tests/Unit/CountryControllerTest.php:95
91 Country::create($datum);
92 }
93
94 $response = $this->getJson("/api/v1/countries/AX");
→ 95 $response->assertStatus(200)->assertJson($data);
96 });
97

Tests: 2 failed, 4 passed (9 assertions)
Duration: 0.65s
```

The results for our tests are shown here as plain text:

```
php artisan test

PASS Tests\Feature\BrowseRegionsTest
✓ it can fetch all regions
0.20s

PASS Tests\Feature\ExampleTest
✓ it has welcome page [Feature]
0.03s

FAIL Tests\Feature\GetRegionTest
× it can fetch one Region by ID
0.02s
```

```

FAILED Tests\Feature\GetRegionTest > it can fetch one Region by ID
AssertionFailedError
Invalid JSON was returned from the route.

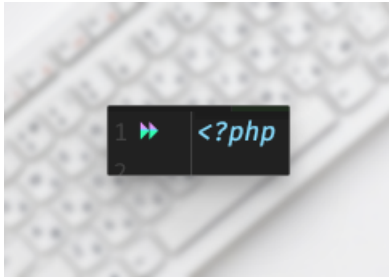
at tests/Feature/GetRegionTest.php:41
 37 |         Region::create($datum);
 38 |     }
 39 |
 40 |     $response = $this->getJson('/api/v1/regions/2');
→ 41 |     $response->assertStatus(200)->assertJson($data);
 42 | });
 43 |

Tests: 1 failed, 2 passed (5 assertions)
Duration: 0.35s
```

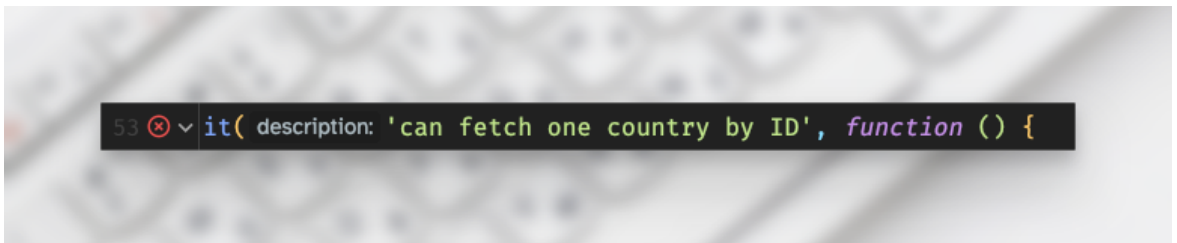
Executing Tests in PhpStorm

It is possible to run your tests in PhpStorm. This is done by:

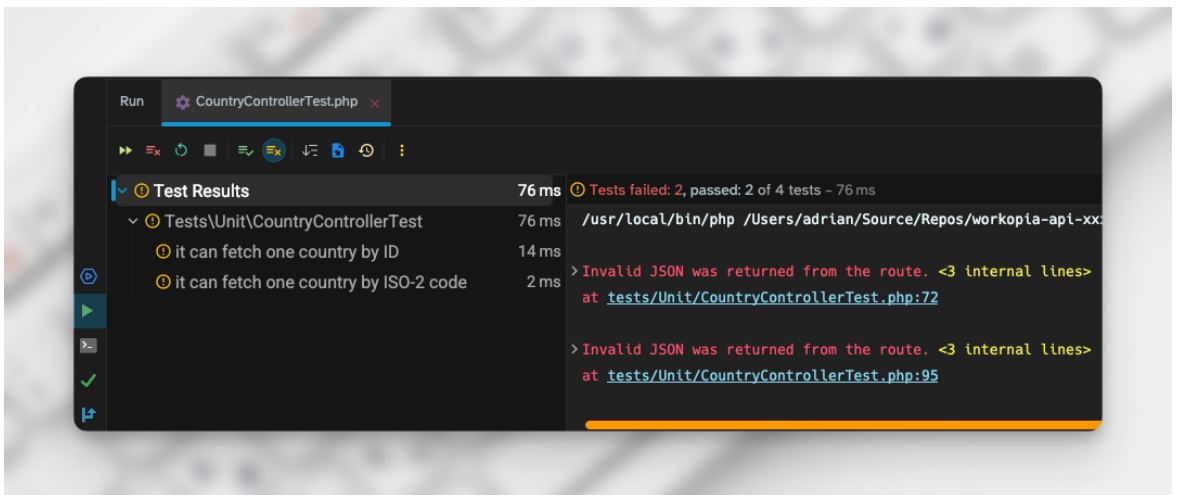
- Open the Test File (Region Controller Test)
- At the top of the file, you will see a double chevron `⌵`:LiChevransRight:, click this to run all tests in the file:



- Alternatively to run a single test, find an arrow `⬆`:LiChevronRightSquare:, cross `⊗`:LiCross: or tick `☑`:LiCheck: next to the test and click it (screenshot for illustration only):



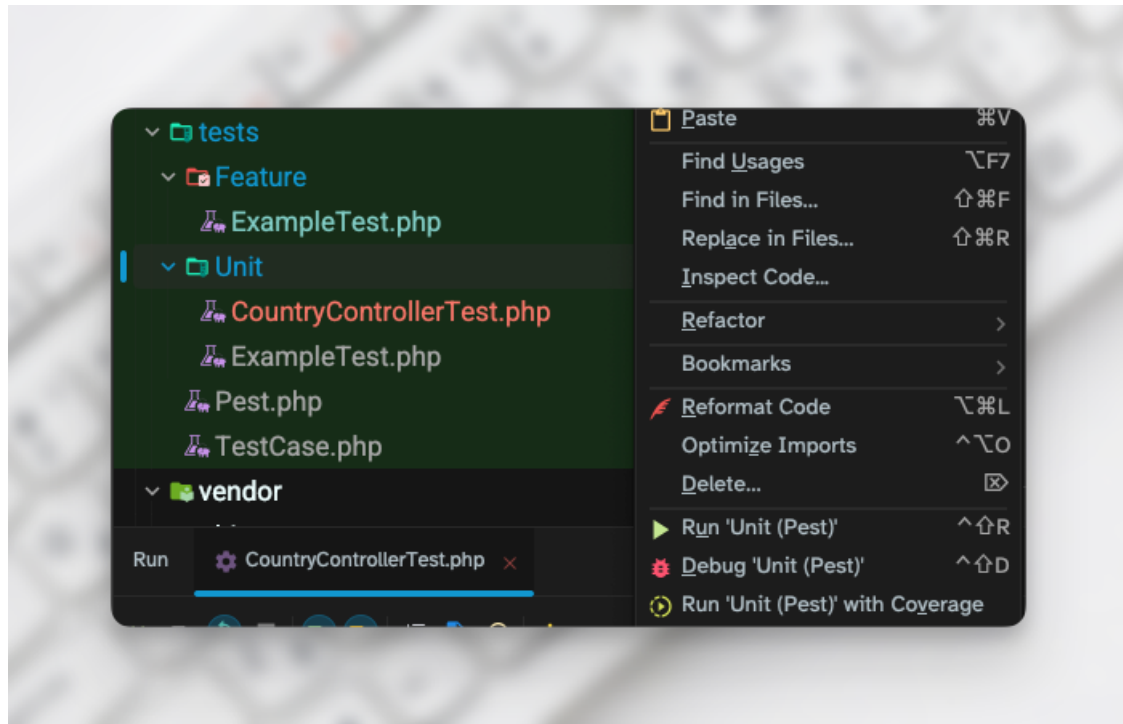
- The test is executed and details shown in the run pane (screenshot for illustration only):



The advantage of this is that you can click on the error line in the right side to jump straight to the line with the issue. In this case the code in the test.

Alternative Ways To Run Tests in PhpStorm

You can also run all tests in PhpStorm by right clicking on the `tests` folder and selecting the relevant Run ... (Pest) option. For example, right clicking on the `Unit` folder will show "Run Unit (Pest)". Doing the same on the `tests` folder will allow all tests to execute.



3) Write Code

Identify the issue to write the code for, and write the code to solve the problem.

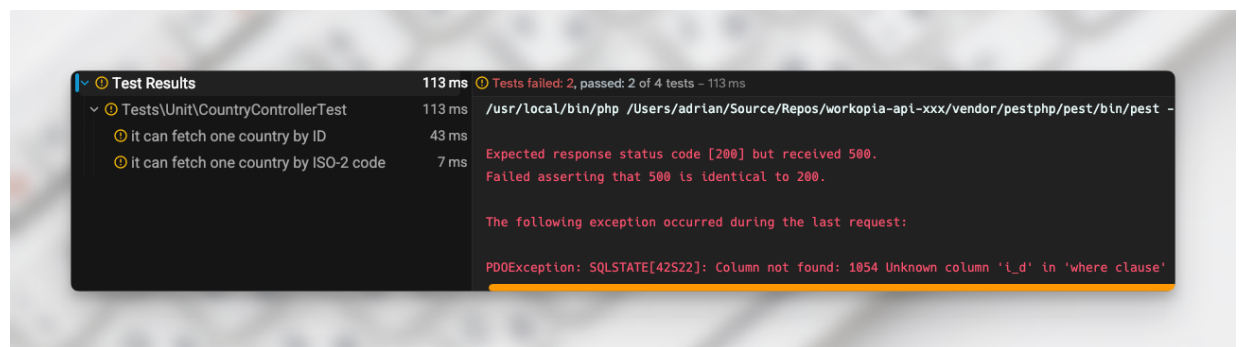
Let's begin with retrieving by ID:

```
public function show(string $id)
{
    $Region = Region::whereID($id)->get();
    $message = count($Region) > 0 ? "Region Found" : "Region Not Found";
    return ApiResponseClass::sendResponse($Region, $message);
}
```

4) Rerun Tests

Using your preferred method, re-run the tests.

Here we see a problem:



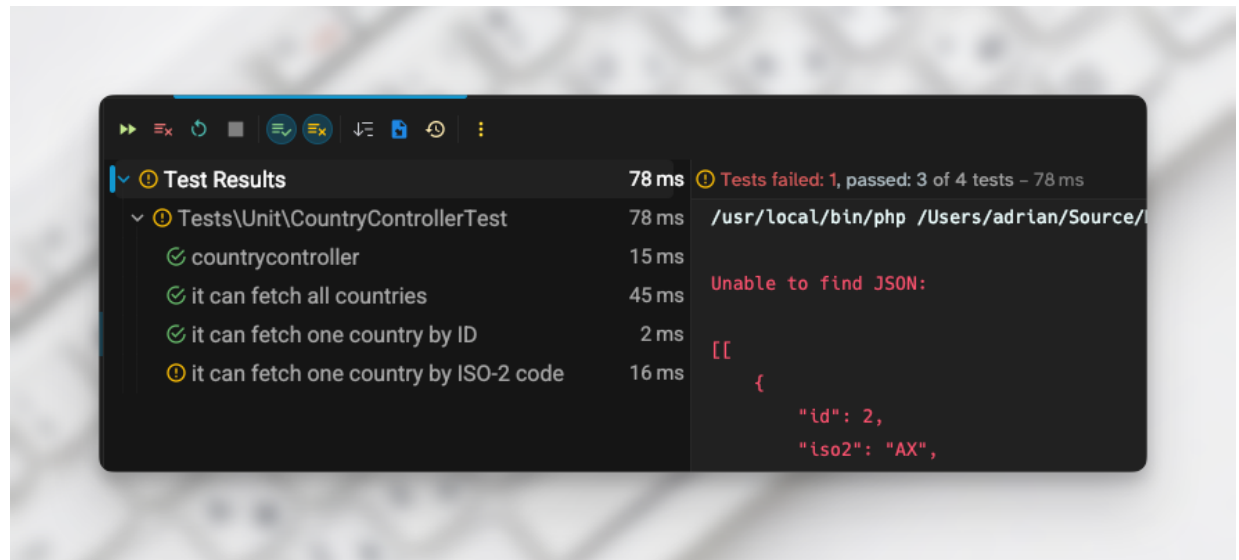
This is a common mistake... the `whereID` needs to adhere to the Camel case rules... `whereId`.

5) Fix the error and go back to step 4

Fix the issue, and discussed in the previous step, and re-run the tests.

```
public function show(string $id)
{
    $Region = Region::whereId((int)$id)->get();
    $message = count($Region) > 0 ? "Region Found" : "Region Not Found";
    return ApiResponseClass::sendResponse($Region, $message);
}
```

Here is the result:



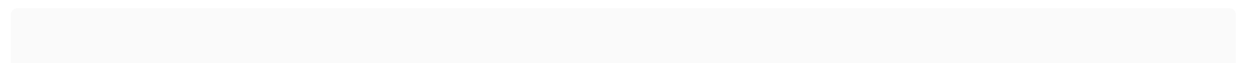
:LiAlertTriangle: **Note:** We have introduced a possible issue by forcing the type casting on `$id`.

6) Refactor Code

Once you have done any refactoring, you **must** re-run the tests to ensure you have not introduced any problems whilst refactoring.

7) Document

Do not forget we need to document the API... so edit the controller and add:



If all is well, then you could commit your work, and start on the next part of the development...

What about the Route Refactoring?

We did mention that we could refactor the routes.

In the current code we have:

```
Route::group(['prefix' => 'v1'], function () {
    Route::get('regions', [RegionController::class, 'index'])->name('region.index');
    Route::get('regions/{id}', [RegionController::class, 'show'])->name('region.show');
});
```

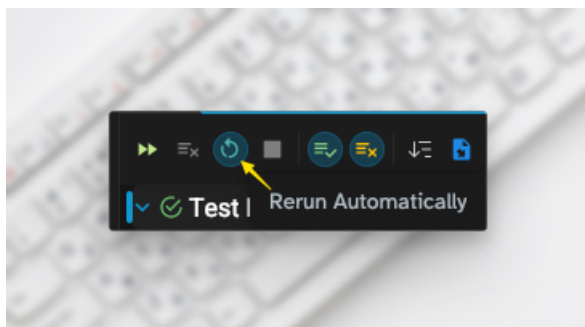
It is possible to use the Resourceful API routing to do the same, but tell it to `only` work for the index and show methods:

```
Route::group(['prefix' => 'v1'], function () {
    Route::apiResource('regions', RegionController::class)
        ->only(['index', 'show']);
});
```

What about Automatic Test Execution?

We can do this in PhpStorm by using the Rerun Automatically option.

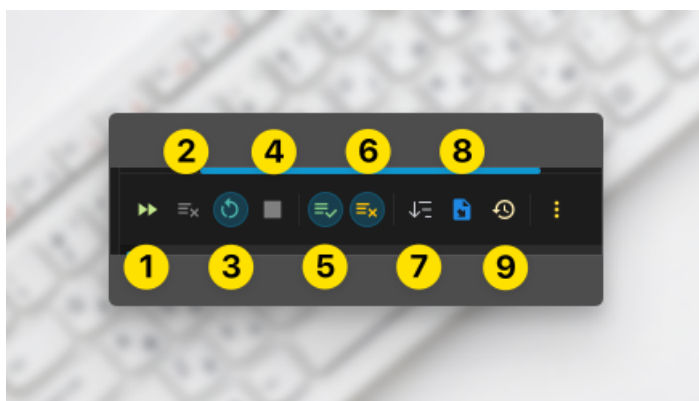
Find the button with a 'circular' arrow and click this.



You should now have automatic re-running of tests.

PhpStorm Test Runner Icons

To quickly assist you locate the correct icons in the test runner toolbar, here is the toolbar and the icons.



1. Run test(s)
2. Rerun failed tests

3. Automatic rerun
 4. Stop running tests
 5. Show passed
 6. Show Failed
 7. Sorting options
 8. Import tests from file
 9. Test History
-

Exercises

The following are exercises for you to use to add features to the API as practice.

Exercise 1: Subregion

We have completed the Region, next we need to add the Subregion

Subregion model details

When you complete the same as we have done for the Region, make sure that the Subregion model contains only the following:

- id
- name
- region_id

Make sure you write the tests to go with the index and show methods.

The region id should not be string, but an unsigned integer.

Add Documentation

Add the required Php DocBlocks to the methods in the Controller to allow Scribe to autogenerate the documents.

These methods should be in the "Subregions" group.

Exercise 2: Country

We have completed the Region and Subregion, next we need to add the Country.

Country model details

When you complete the same as we have done for the Region, make sure that the Country model contains only the following:

- name
- iso3
- iso2
- numeric_code
- phone_code
- capital
- currency
- currency_name
- currency_symbol
- tld
- region
- region_id
- subregion

- subregion_id
- emoji

Make sure you write the tests to go with the index and show methods.

The region id, subregion id, and numeric code should not be string, but unsigned integers.

Add Documentation

Add the required Php DocBlocks to the methods in the Controller to allow Scribe to autogenerate the documents.

These methods should be in the "Countries" group.

Exercise 3: States

Let's keep this going and create the States API...

State model details

When you complete the same as we have done for the Region, make sure that the States model contains only the following:

- id
- name
- country_id
- country_code
- country_name
- state_code
- type

Make sure you write the tests to go with the index and show methods.

Make sure that the country code and country id in this model matches the data types in the Country model. The state code is a string, max length of 6 characters.

Add Documentation

Add the required Php DocBlocks to the methods in the Controller to allow Scribe to autogenerate the documents.

These methods should be in the "States" group.

Exercise 4: Cities

On the last leg for this, and it's the biggest of them all. The Cities!

City model details

Following the same steps as before, add only the following:

- id

- name
- state_id
- state_code
- state_name
- country_id
- country_code
- country_name

Make sure you write the tests to go with the index and show methods.

Make sure that the country code and country id in this model matches the data types in the Country model.

Add Documentation

Add the required Php DocBlocks to the methods in the Controller to allow Scribe to autogenerate the documents.

These methods should be in the "Cities" group.

Questions to Investigate

You DO NOT need to provide formal answers to these questions. They are a combination of research, review, and thought provoking ideas that may or may not be relevant to the development of an API.

Question: Generalised Response Class - Could we add a "success" field set to false? In either case why?

Question: API Code Reuse - Consider how you may be able to use the API codebase to help develop a web application so that code would not be duplicated.

Question: How could you use GitHub workflows to automatically generate the Scribe documentation on push/pull request?

Question: How could you write the code in your controller methods so that if nothing was retrieved, you returned a 404 not found response? (ties in with previous question)

END

Next up: [Our First API](#)