Our-First-API

TDD & Our First API

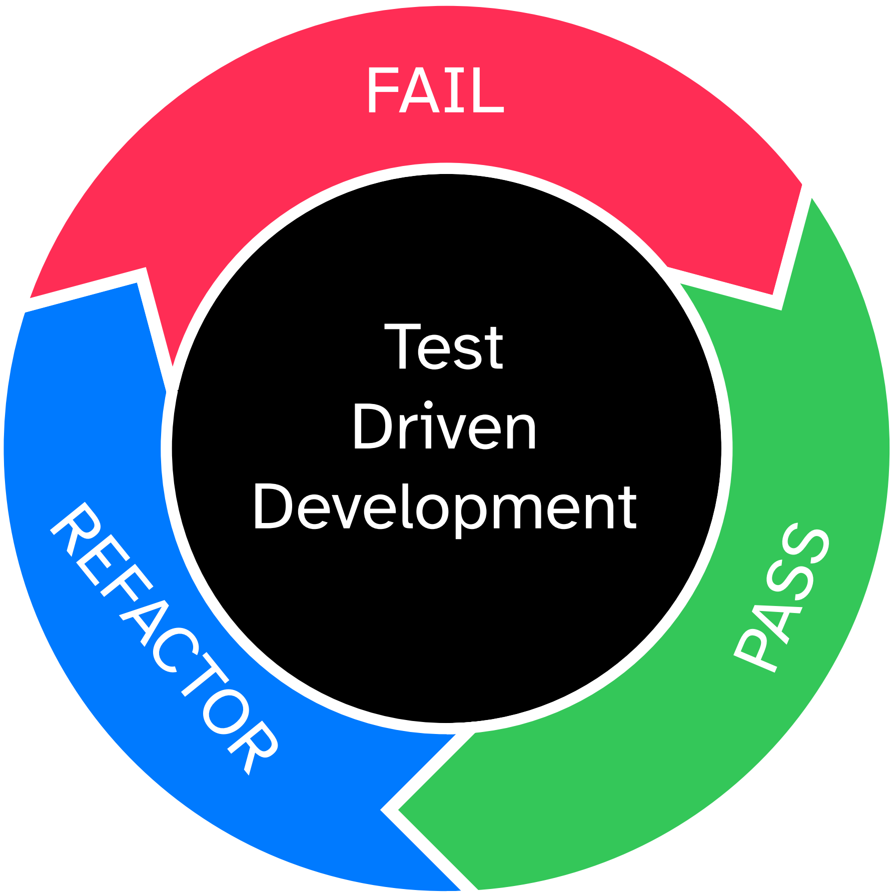
Software as a Service - Back-End Development

Session 01

Developed by Adrian Gould

Contents

Test Driven Development



Test Driven Development is a methodology that works on the principle that you write tests, run the tests to fail, write code to fix the fail, rerun the tests to pass, refactor your code (testing continuously), and repeat.

Each loop will, be for a *single* feature or sub-feature.

For example, when developing an API feature that will perform CRUD/BREAD operations it is a good principle break this into the separate steps:

- Browse
- Read
- Edit
- Add
- Delete

We will often perform the creation of these sub-features in Browse, Read, Add, Edit Delete order.

Company API

To show how we develop a complete feature, we will use the Company model as the platform to spring from.

Company Feature User Stories

For these features we will start with no security. Later we will revisit and add security so that only registered users and site staff can add/edit/delete companies.

The two basic stories that do not manipulate data:

- As a user I must be able to list all companies
- As a user I must be able to show a company's details
  The three stories that manipulate the data:
- As a user I must be able to add a company
- As a user I must be able to edit a company
- As a user I must be able to delete a company
  The extension to the list all to enable searching:
- As a user I must be able to search for a company

Data Storage

One of the issues we could have with this particular model is how we are storing the locations for the companies.

Ideally we would have a pivot table containing company locations and use a location model to provide the city, region and country so we do not have to repeat the data.

In this example, though, we will be storing these items of data with the company, and use Country, City, Region and other tables as lookups for filling in the data in the Company model.

For the purpose of this example, we will fill the data in by hand and use the faking capabilities from Faker.

Future enhancement (refactoring?) could use the models described when doing the inserts, as the front end would determine where the data came from (usually some form of lookup and autocomplete).

As a user I must be able to list all companies

Write Test

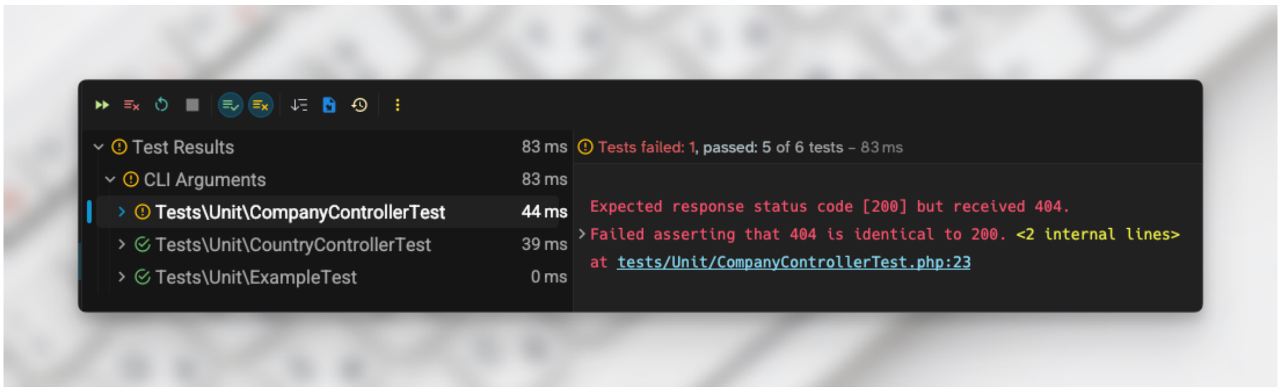
```
php artisan make:test --pest --unit CompanyControllerTest
```

Open the Test file, remove the sample test and add the first of the tests.

```php
<?php

use App\Models\Company;
use Illuminate\Foundation\Testing\RefreshDatabase;

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

it('can fetch all countries', function () {
    $companies = Company::factory(5)->create();

    $data = [
        'message' => 'Companies retrieved successfully.',
        "success" => true,
        'data' => $companies->toArray(),
    ];

    $response = $this->getJson("/api/v1/companies");
    $response→assertStatus(200)→assertJson($data);
});
```

Run Test

Executing the test will fail...



```
Test Results                          83 ms   Tests failed: 1, passed: 5 of 6 tests – 83 ms
  CLI Arguments                       83 ms
    Tests\Unit\CompanyControllerTest  44 ms   Expected response status code [200] but received 404.
    Tests\Unit\CountryControllerTest  39 ms   Failed asserting that 404 is identical to 200. <2 internal lines>
    Tests\Unit\ExampleTest             0 ms   at tests/Unit/CompanyControllerTest.php:23
```

So we now know we do not have a route...

Create Route

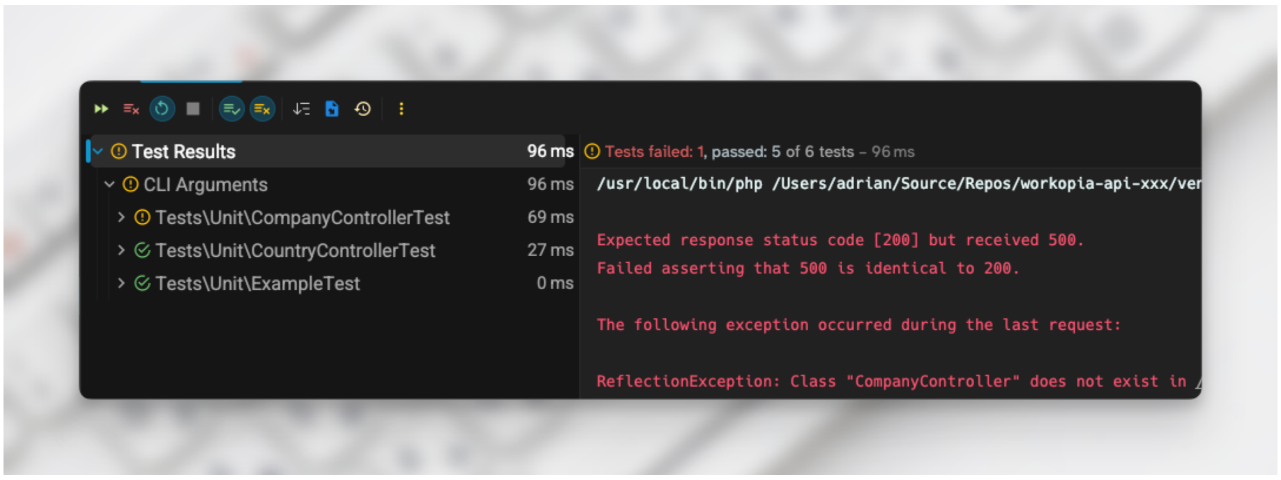
OK, so we now know we cannot find the described route, so let's add an API resourceful route.

Edit the `routes/api.php` file, and add the companies API resource route that calls CompanyController

```
Route::group(['prefix' ⇒ 'v1'], function () {
    Route::apiResource('/companies', CompanyController::class);
    Route::apiResource('/countries', CountryController::class);
});
```

Run Test

Executing the test will fail...



```
Test Results                          96 ms   Tests failed: 1, passed: 5 of 6 tests – 96 ms
  CLI Arguments                       96 ms   /usr/local/bin/php /Users/adrian/Source/Repos/workopia-api-xxx/ver
    Tests\Unit\CompanyControllerTest  69 ms
    Tests\Unit\CountryControllerTest  27 ms   Expected response status code [200] but received 500.
    Tests\Unit\ExampleTest             0 ms   Failed asserting that 500 is identical to 200.

                                              The following exception occurred during the last request:

                                              ReflectionException: Class "CompanyController" does not exist in
```

We can now see we do not have the controller...

Create API Controller

Create the Controller Stub:

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

Run Test

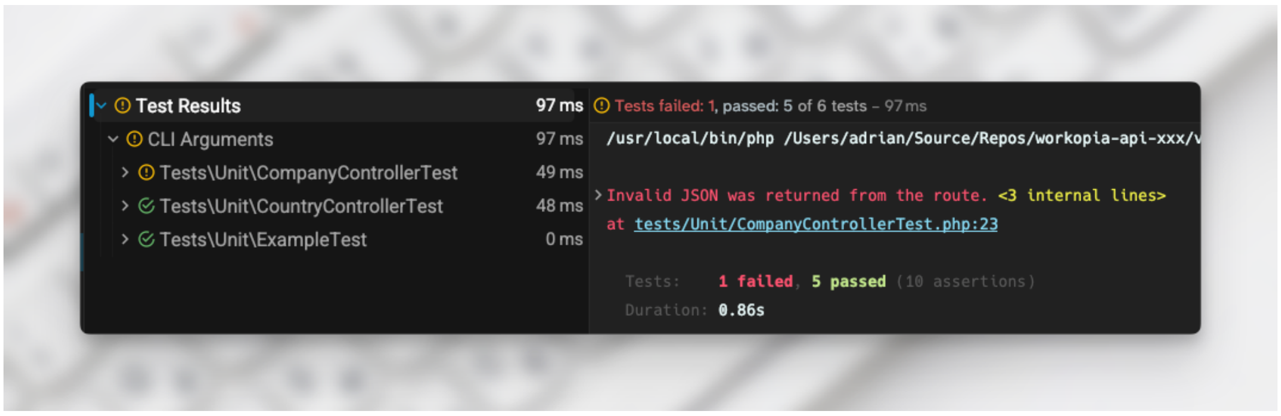
Executing the test will fail as we have not added teh Company Controller to the test file.

Edit the CompanyControllerTest.php file and add the following use line at the top of the file after the `<?php` tag:

```
use App\Http\Controllers\CompanyController;
```

Re-run the tests

Excellent! We have got somewhere... even if we have a different problem to solve.



```
Test Results                                97 ms
  CLI Arguments                             97 ms
    Tests\Unit\CompanyControllerTest        49 ms
    Tests\Unit\CountryControllerTest        48 ms
    Tests\Unit\ExampleTest                   0 ms
```

```
Tests failed: 1, passed: 5 of 6 tests – 97 ms
/usr/local/bin/php /Users/adrian/Source/Repos/workopia-api-xxx/v

> Invalid JSON was returned from the route. <3 internal lines>
  at tests/Unit/CompanyControllerTest.php:23

   Tests:     1 failed, 5 passed (10 assertions)
   Duration: 0.86s
```

Create Migration

At this point we are going to create the migration for the company model.

Important: remember that the table names are PLURALISED versions of the model names. For example:

Model Name	Table Name
Book	books
Country	countries
Property	properties
Company	companies
Fish	fish

_Aside: sometimes plurals may not follow the expected rules, but we know we can create models, migrations, factories and seeders in one go using `php artisan make:model Fish --migration --seeder --factory` or in short hand `php artisan make:model Fish -m -s -f`.

```
php artisan make:migration create_companies_table
```

In the new migration we add the company table specifications between the `id` and `timestamps` ...

```
$table→string('name',192);
$table→string('city',192)→nullable();
$table→string('region', 192)→nullable();
$table→string('country', 192)→default('AU');
$table→string('url')→nullable();
```

For this example we are going to add full city, region (state) and country to the table, even though we are duplicating the data.

We also need to add a way to make the city, region, country and company name unique, so we will combine these fields together to create a unique index.

There is an issue that if we do not limit the lengths then the index we will need to create will cause an error... so we have limited the lengths of the properties to 192 characters for now.

We can do some exploration and alter this later.

Anyway, we can now add the unique index creation just after the `timestamps()` line we will add a blank line and then...

```
$table->unique(['name', 'city', 'region', 'country'], 'city_location_index');
```

Run the migration

Run the migration using:

```
php artisan migrate
```

Create Factory

Now we have the migration, we add a CompanyFactory...

```
php artisan make:factory CompanyFactory
```

In the new factory add the following to the `return` statement:

```
'name' => $this->faker->company(),
'city' => $this->faker->city(),
'region' => $this->faker->word(1),
'country' => $this->faker->country(),
'url' => $this->faker->url(),
```

It should be pretty easy to see what data is being faked.

Create Model

Next we get to the Company Model...

```
php artisan make:model Company
```

Edit the model and add the following immediately after the `use HasFactory` line:

```
/**
 * The attributes that will be mass assignable for Country. * * @var string[]
 */protected $fillable = [
    'name',
    'city',
    'region',
    'country',
    'url',
];

/**
 * 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 [
    ];
}
```

At this point we are ready to start coding our controller.

Create the Company Controller index method

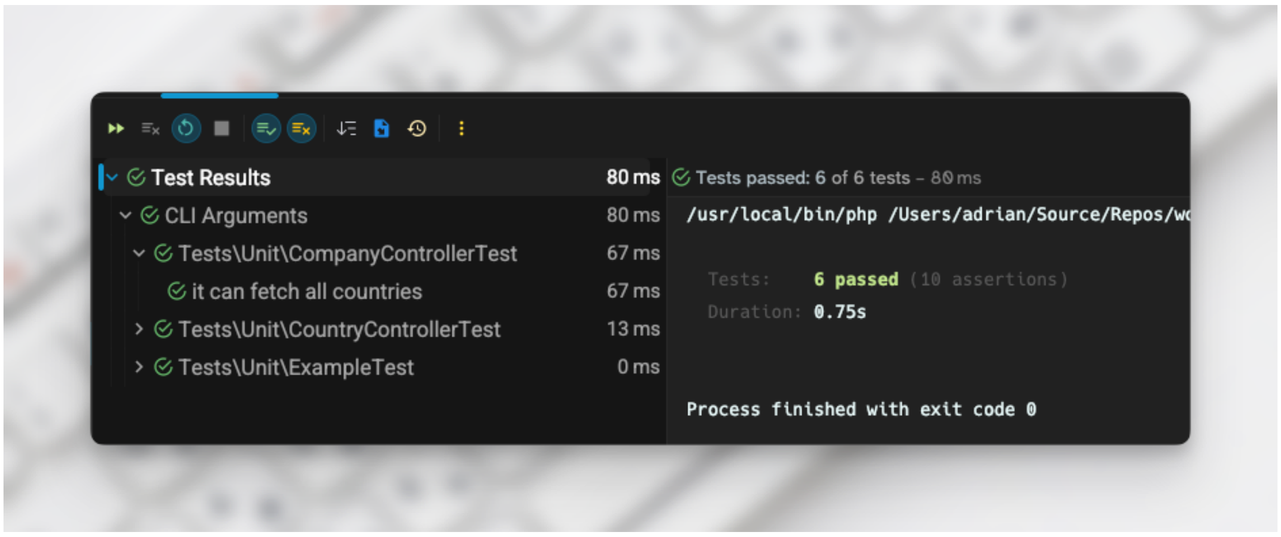
Open the `CompanyController` and find the `index()` method. Edit it to read:

```php
$companies  = Company::all();

return ApiResponseClass::sendResponse($companies, "Companies retrieved successfully.", 200);
```

Remember you will have to add the relevant `use` lines.

Run Test

Does it work? If all is coded correctly then we will get 6 tests passed.



```
Test Results                                  80 ms
  CLI Arguments                               80 ms
    Tests\Unit\CompanyControllerTest          67 ms
      it can fetch all countries              67 ms
    Tests\Unit\CountryControllerTest          13 ms
    Tests\Unit\ExampleTest                     0 ms
```

```
Tests passed: 6 of 6 tests – 80 ms
/usr/local/bin/php /Users/adrian/Source/Repos/w

  Tests:    6 passed (10 assertions)
  Duration: 0.75s


Process finished with exit code 0
```

Success! :FasCheckDouble:

Refactor

This code does not need refactoring, so we shall progress onto the Show method.

As a user I must be able to show a company

Let us repeat this process:

1. Write Test
2. Fail Test
3. Write Code
4. Pass Test
5. Refactor
6. Repeat

One thing to note, is that if a feature requires more complex behaviours, then there is no problem with repeating steps 1 to 5 until all the requirements for that feature are met.

Write Test

The "" test will read something similar to:

Fail Test

Running the test should give us a failure.

Write Code

We now write the code to pass the test as it stands...

Pass Test

Re-running the test, should allow for the pass to be present. If not then go back and check and write code (DO NOT ALTER THE TESTS!)...

Refactor

If you have a verbose method, then it may be a prudent action to refactor to make it more efficient.

This will depend on the situation.

Rinse and Repeat

We are not needing any extra work with this particular method.

As a user I must be able to create a Company

OK, we are now onto the first of the methods that change the database.

We will follow the same steps as before.

Write Test

The "" test will read something similar to:

Fail Test

Running the test should give us a failure.

Write Code

We now write the code to pass the test as it stands...

Pass Test

Re-running the test, should allow for the pass to be present. If not then go back and check and write code (DO NOT ALTER THE TESTS!)...

Refactor

If you have a verbose method, then it may be a prudent action to refactor to make it more efficient.

This will depend on the situation.

Rinse and Repeat

We are not needing any extra work with this particular method.

As a user I must be able to delete a company

This method may cause some consternation as to if we should allow for company details to be removed.

The problem will be what do we do with any data associated with that company? In many situations we would probably say that company names and locations will not be allowed to be deleted, but they may be soft deleted, along with records that relate to them. The soft delete would allow the site to still perform some data analysis if needed.

For the purpose of this exercise we will create the delete() method, and test it.

Write Test

The "" test will read something similar to:

```
```

Fail Test

Running the test should give us a failure.

Write Code

We now write the code to pass the test as it stands…

```
```

Pass Test

Re-running the test, should allow for the pass to be present. If not then go back and check and write code (DO NOT ALTER THE TESTS!)…

Refactor

If you have a verbose method, then it may be a prudent action to refactor to make it more efficient.

This will depend on the situation.

Rinse and Repeat

We are not needing any extra work with this particular method.

As a user I must be able to edit a company

Edit should be a good exercise in determining how to update the the company details. The way we have designed this particular system, a company would be added multiple times if they have multiple locations. Not the most efficient and definitely not a normalised database design.

Nonetheless we will implement the edit.

Write Test

The "" test will read something similar to:

Fail Test

Running the test should give us a failure.

Write Code

We now write the code to pass the test as it stands...

Pass Test

Re-running the test, should allow for the pass to be present. If not then go back and check and write code (DO NOT ALTER THE TESTS!)...

Refactor

If you have a verbose method, then it may be a prudent action to refactor to make it more efficient.

This will depend on the situation.

Rinse and Repeat

We are not needing any extra work with this particular method.

To be continued

END

Next Up: Journals