Testing

- # Introduction
- # Application Testing
  - # Interacting With Your Application
  - # Testing JSON APIs
  - # Sessions / Authentication
  - # Disabling Middleware
  - # Custom HTTP Requests
  - # PHPUnit Assertions
- # Working With Databases
  - # Resetting The Database After Each Test
  - # Model Factories
- # Mocking
  - # Mocking Events
  - # Mocking Jobs
  - # Mocking Facades

Introduction

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included out of the box, and a `phpunit.xml` file is already setup for your application. The framework also ships with convenient helper methods allowing you to expressively test your applications.

An `ExampleTest.php` file is provided in the `tests` directory. After installing a new Laravel application, simply run `phpunit` on the command line to run your tests.

Test Environment

When running tests, Laravel will automatically set the configuration environment to `testing`. Laravel automatically configures the session and cache to the `array` driver while testing, meaning no session or cache data will be persisted while testing.

You are free to create other testing environment configurations as necessary. The `testing` environment variables may be configured in the `phpunit.xml` file.

Defining & Running Tests

To create a new test case, use the `make:test` Artisan command:

```
php artisan make:test UserTest
```

This command will place a new `UserTest` class within your `tests` directory. You may then define test methods as you normally would using PHPUnit. To run your tests, simply execute the `phpunit` command from your terminal:

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class UserTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testExample()
    {
        $this->assertTrue(true);
    }
}
```

Note: If you define your own `setUp` method within a test class, be sure to call `parent::setUp`.

Application Testing

Laravel provides a very fluent API for making HTTP requests to your application, examining the output, and even filling out forms. For example, take a look at the `ExampleTest.php` file included in your `tests` directory:

```php
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5')
             ->dontSee('Rails');
    }
}
```

The `visit` method makes a `GET` request into the application. The `see` method asserts that we should see the given text in the response returned by the application. The `dontSee` method asserts that the given text is not returned in the application response. This is the most basic application test available in Laravel.

Interacting With Your Application

Of course, you can do much more than simply assert that text appears in a given response. Let's take a look at some examples of clicking links and filling out forms:

Clicking Links

In this test, we will make a request to the application, "click" a link in the returned response, and then assert that we landed on a given URI. For example, let's assume there is a link in our response that has a text value of "About Us":

```php
<a href="/about-us">About Us</a>
```

Now, let's write a test that clicks the link and asserts the user lands on the correct page:

```php
public function testBasicExample()
{
    $this->visit('/')
         ->click('About Us')
         ->seePageIs('/about-us');
}
```

Working With Forms

Laravel also provides several methods for testing forms. The `type`, `select`, `check`, `attach`, and `press` methods allow you to interact with all of your form's inputs. For example, let's imagine this form exists on the application's registration page:

```
<form action="/register" method="POST">
    {!! csrf_field() !!}

    <div>
        Name: <input type="text" name="name">
    </div>

    <div>
        <input type="checkbox" value="yes" name="terms"> Accept Terms
    </div>

    <div>
        <input type="submit" value="Register">
    </div>
</form>
```

We can write a test to complete this form and inspect the result:

```
public function testNewUserRegistration()
{
    $this->visit('/register')
         ->type('Taylor', 'name')
         ->check('terms')
         ->press('Register')
         ->seePageIs('/dashboard');
}
```

Of course, if your form contains other inputs such as radio buttons or drop-down boxes, you may easily fill out those types of fields as well. Here is a list of each form manipulation method:

Method	Description
`$this->type($text, $elementName)`	"Type" text into a given field.
`$this->select($value, $elementName)`	"Select" a radio button or drop-down field.
`$this->check($elementName)`	"Check" a checkbox field.
`$this->attach($pathToFile, $elementName)`	"Attach" a file to the form.
`$this->press($buttonTextOrElementName)`	"Press" a button with the given text or name.

Working With Attachments

If your form contains `file` input types, you may attach files to the form using the `attach` method:

```
public function testPhotoCanBeUploaded()
{
    $this->visit('/upload')
        ->name('File Name', 'name')
        ->attach($absolutePathToFile, 'photo')
        ->press('Upload')
        ->see('Upload Successful!');
}
```

Testing JSON APIs

Laravel also provides several helpers for testing JSON APIs and their responses. For example, the `get`, `post`, `put`, `patch`, and `delete` methods may be used to issue requests with various HTTP verbs. You may also easily pass data and headers to these methods. To get started, let's write a test to make a `POST` request to `/user` and assert that a given array was returned in JSON format:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJson([
                'created' => true,
             ]);
    }
}
```

The `seeJson` method converts the given array into JSON, and then verifies that the JSON fragment occurs **anywhere** within the entire JSON response returned by the application. So, if there are other properties in the JSON response, this test will still pass as long as the given fragment is present.

Verify Exact JSON Match

If you would like to verify that the given array is an **exact** match for the JSON returned by the application, you should use the `seeJsonEquals` method:

```php
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJsonEquals([
                 'created' => true,
             ]);
    }
}
```

Sessions / Authentication

Laravel provides several helpers for working with the session during testing. First, you may set the session data to a given array using the `withSession` method. This is useful for loading the session with data before testing a request to your application:

```php
<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $this->withSession(['foo' => 'bar'])
             ->visit('/');
    }
}
```

Of course, one common use of the session is for maintaining user state, such as the authenticated user. The `actingAs` helper method provides a simple way to authenticate a given user as the current user. For example, we may use a model factory to generate and authenticate a user:

```php
<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $user = factory(App\User::class)->create();

        $this->actingAs($user)
             ->withSession(['foo' => 'bar'])
             ->visit('/')
             ->see('Hello, '.$user->name);
    }
}
```

Disabling Middleware

When testing your application, you may find it convenient to disable middleware for some of your tests. This will allow you to test your routes and controller in isolation from any middleware concerns. Laravel includes a simple `WithoutMiddleware` trait that you can use to automatically disable all middleware for the test class:

```php
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use WithoutMiddleware;

    //
}
```

If you would like to only disable middleware for a few test methods, you may call the `withoutMiddleware` method from within the test methods:

```php
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->withoutMiddleware();

        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

Custom HTTP Requests

If you would like to make a custom HTTP request into your application and get the full `Illuminate\Http\Response` object, you may use the `call` method:

```php
public function testApplication()
{
    $response = $this->call('GET', '/');

    $this->assertEquals(200, $response->status());
}
```

If you are making `POST`, `PUT`, or `PATCH` requests you may pass an array of input data with the request. Of course, this data will be available in your routes and controller via the Request instance:

```php
$response = $this->call('POST', '/user', ['name' => 'Taylor']);
```

PHPUnit Assertions

Laravel provides several additional assertion methods for PHPUnit tests:

Method	Description
`->assertResponseOk();`	Assert that the client response has an OK status code.
`->assertResponseStatus($code);`	Assert that the client response has a given code.
`->assertViewHas($key, $value = null);`	Assert that the response view has a given piece of bound data.
`->assertViewHasAll(array $bindings);`	Assert that the view has a given list of bound data.

Method	Description
`->assertViewMissing($key);`	Assert that the response view is missing a piece of bound data.
`->assertRedirectedTo($uri, $with = []);`	Assert whether the client was redirected to a given URI.
`->assertRedirectedToRoute($name, $parameters = [], $with = []);`	Assert whether the client was redirected to a given route.
`->assertRedirectedToAction($name, $parameters = [], $with = []);`	Assert whether the client was redirected to a given action.
`->assertSessionHas($key, $value = null);`	Assert that the session has a given value.
`->assertSessionHasAll(array $bindings);`	Assert that the session has a given list of values.
`->assertSessionHasErrors($bindings = [], $format = null);`	Assert that the session has errors bound.
`->assertHasOldInput();`	Assert that the session has old input.

Working With Databases

Laravel also provides a variety of helpful tools to make it easier to test your database driven applications. First, you may use the `seeInDatabase` helper to assert that data exists in the database matching a given set of criteria. For example, if we would like to verify that there is a record in the `users` table with the `email` value of `sally@example.com`, we can do the following:

```
public function testDatabase()
{
    // Make call to application...

    $this->seeInDatabase('users', ['email' => 'sally@example.com']);
}
```

Of course, the `seeInDatabase` method and other helpers like it are for convenience. You are free to use any of PHPUnit's built-in assertion methods to supplement your tests.

Resetting The Database After Each Test

It is often useful to reset your database after each test so that data from a previous test does not interfere with subsequent tests.

Using Migrations

One option is to rollback the database after each test and migrate it before the next test. Laravel provides a simple `DatabaseMigrations` trait that will automatically handle this for you. Simply use the trait on your test class:

```php
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

Using Transactions

Another option is to wrap every test case in a database transaction. Again, Laravel provides a convenient `DatabaseTransactions` trait that will automatically handle this:

```php
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseTransactions;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

Note: This trait will only wrap the default database connection in a transaction.

Model Factories

When testing, it is common to need to insert a few records into your database before executing your test. Instead of manually specifying the value of each column when you create this test data, Laravel allows you to define a default set of attributes for each of your Eloquent models using "factories". To get started, take a look at the `database/factories/ModelFactory.php` file in your application. Out of the box, this file contains one factory definition:

```
$factory->define(App\User::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => bcrypt(str_random(10)),
        'remember_token' => str_random(10),
    ];
});
```

Within the Closure, which serves as the factory definition, you may return the default test values of all attributes on the model. The Closure will receive an instance of the Faker PHP library, which allows you to conveniently generate various kinds of random data for testing.

Of course, you are free to add your own additional factories to the `ModelFactory.php` file.

Multiple Factory Types

Sometimes you may wish to have multiple factories for the same Eloquent model class. For example, perhaps you would like to have a factory for "Administrator" users in addition to normal users. You may define these factories using the `defineAs` method:

```
$factory->defineAs(App\User::class, 'admin', function ($faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => str_random(10),
        'remember_token' => str_random(10),
        'admin' => true,
    ];
});
```

Instead of duplicating all of the attributes from your base user factory, you may use the `raw` method to retrieve the base attributes. Once you have the attributes, simply supplement them with any additional values you require:

```
$factory->defineAs(App\User::class, 'admin', function ($faker) use ($factory) {
    $user = $factory->raw(App\User::class);

    return array_merge($user, ['admin' => true]);
});
```

Using Factories In Tests

Once you have defined your factories, you may use them in your tests or database seed files to generate model instances using the global `factory` function. So, let's take a look at a few examples of creating models. First, we'll use the `make` method, which creates models but does not save them to the database:

```
public function testDatabase()
{
    $user = factory(App\User::class)->make();

    // Use model in tests...
}
```

If you would like to override some of the default values of your models, you may pass an array of values to the `make` method. Only the specified values will be replaced while the rest of the values remain set to their default values as specified by the factory:

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
]);
```

You may also create a Collection of many models or create models of a given type:

```
// Create three App\User instances...
$users = factory(App\User::class, 3)->make();

// Create an App\User "admin" instance...
$user = factory(App\User::class, 'admin')->make();

// Create three App\User "admin" instances...
$users = factory(App\User::class, 'admin', 3)->make();
```

Persisting Factory Models

The `create` method not only creates the model instances, but also saves them to the database using Eloquent's `save` method:

```
public function testDatabase()
{
    $user = factory(App\User::class)->create();

    // Use model in tests...
}
```

Again, you may override attributes on the model by passing an array to the `create` method:

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

Adding Relations To Models

You may even persist multiple models to the database. In this example, we'll even attach a relation to the created models. When using the `create` method to create multiple models, an Eloquent collection instance is returned, allowing you to use any of the convenient functions provided by the collection, such as `each`:

```
$users = factory(App\User::class, 3)
        ->create()
        ->each(function($u) {
            $u->posts()->save(factory(App\Post::class)->make());
        });
```

Mocking

Mocking Events

If you are making heavy use of Laravel's event system, you may wish to silence or mock certain events while testing. For example, if you are testing user registration, you probably do not want all of a `UserRegistered` event's handlers firing, since these may send "welcome" e-mails, etc.

Laravel provides a convenient `expectsEvents` method that verifies the expected events are fired, but prevents any handlers for those events from running:

```
<?php

class ExampleTest extends TestCase
{
    public function testUserRegistration()
    {
        $this->expectsEvents(App\Events\UserRegistered::class);

        // Test user registration code...
    }
}
```

If you would like to prevent all event handlers from running, you may use the `withoutEvents` method:

```
<?php

class ExampleTest extends TestCase
{
    public function testUserRegistration()
    {
        $this->withoutEvents();

        // Test user registration code...
    }
}
```

Mocking Jobs

Sometimes, you may wish to simply test that specific jobs are dispatched by your controllers when making requests to your application. This allows you to test your routes / controllers in isolation - set apart from your job's logic. Of course, you can then test the job itself in a separate

test class.

Laravel provides a convenient `expectsJobs` method that will verify that the expected jobs are dispatched, but the job itself will not be executed:

```
<?php

class ExampleTest extends TestCase
{
    public function testPurchasePodcast()
    {
        $this->expectsJobs(App\Jobs\PurchasePodcast::class);

        // Test purchase podcast code...
    }
}
```

Note: This method only detects jobs that are dispatched via the `DispatchesJobs` trait's dispatch methods. It does not detect jobs that are sent directly to `Queue::push`.

Mocking Facades

When testing, you may often want to mock a call to a Laravel facade. For example, consider the following controller action:

```
<?php

namespace App\Http\Controllers;

use Cache;
use Illuminate\Routing\Controller;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

We can mock the call to the `Cache` facade by using the `shouldReceive` method, which will return an instance of a Mockery mock. Since facades are actually resolved and managed by the Laravel service container, they have much more testability than a typical static class. For example, let's mock our call to the `Cache` facade:

```php
<?php

class FooTest extends TestCase
{
    public function testGetIndex()
    {
        Cache::shouldReceive('get')
                    ->once()
                    ->with('key')
                    ->andReturn('value');

        $this->visit('/users')->see('value');
    }
}
```

Note: You should not mock the `Request` facade. Instead, pass the input you desire into the HTTP helper methods such as `call` and `post` when running your test.