Prologue
Getting Started
Architecture Concepts
The Basics
Frontend
Security
Digging Deeper
Database
Eloquent ORM
Testing
- Getting Started
- HTTP Tests
- Console Tests
- Browser Tests
- Database
- ● Mocking

Official Packages

# Mocking

# Introduction

When testing Laravel applications, you may wish to "mock" certain aspects of your application so they are not actually executed during a given test. For example, when testing a controller that dispatches an event, you may wish to mock the event listeners so they are not actually executed during the test. This allows you to only test the controller's HTTP response without worrying about the execution of the event listeners, since the event listeners can be tested in their own test case.

Laravel provides helpers for mocking events, jobs, and facades out of the box. These helpers primarily provide a convenience layer over Mockery so you do not have to manually make complicated Mockery method calls. You can also use Mockery or PHPUnit to create your own mocks or spies.

Mocking Objects

When mocking an object that is going to be injected into your application via Laravel's service container, you will need to bind your mocked instance into the container as an `instance` binding. This will instruct the container to use your mocked instance of the object instead of constructing the object itself:

```
use App\Service;
use Mockery;

$this->instance(Service::class, Mockery::mock(Service::class, function ($mock) {
    $mock->shouldReceive('process')->once();
}));
```

In order to make this more convenient, you may use the `mock` method, which is provided by Laravel's base test case class:

```
use App\Service;

$this->mock(Service::class, function ($mock) {
    $mock->shouldReceive('process')->once();
});
```

You may use the `partialMock` method when you only need to mock a few methods of an object. The methods that are not mocked will be executed normally when called:

```
use App\Service;

$this->partialMock(Service::class, function ($mock) {
    $mock->shouldReceive('process')->once();
});
```

Similarly, if you want to spy on an object, Laravel's base test case class offers a `spy` method as a convenient wrapper around the `Mockery::spy` method:

```
use App\Service;

$this->spy(Service::class, function ($mock) {
    $mock->shouldHaveReceived('process');
});
```

Bus Fake

As an alternative to mocking, you may use the `Bus` facade's `fake` method to prevent jobs from being dispatched. When using fakes, assertions are made after the code under test is executed:

```php
<?php

namespace Tests\Feature;

use App\Jobs\ShipOrder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Bus;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Bus::fake();

        // Perform order shipping...

        Bus::assertDispatched(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // Assert a job was not dispatched...
        Bus::assertNotDispatched(AnotherJob::class);
    }
}
```

Event Fake

As an alternative to mocking, you may use the `Event` facade's `fake` method to prevent all event listeners from executing. You may then assert that events were dispatched and even inspect the data they received. When using fakes, assertions are made after the code under test is executed:

```php
<?php

namespace Tests\Feature;

use App\Events\OrderFailedToShip;
use App\Events\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Event;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Test order shipping.
     */
    public function testOrderShipping()
    {
        Event::fake();

        // Perform order shipping...

        Event::assertDispatched(OrderShipped::class, function ($e) use ($order) {
            return $e->order->id === $order->id;
        });

        // Assert an event was dispatched twice...
        Event::assertDispatched(OrderShipped::class, 2);

        // Assert an event was not dispatched...
        Event::assertNotDispatched(OrderFailedToShip::class);
    }
}
```

! After calling `Event::fake()`, no event listeners will be executed. So, if your tests use model factories that rely on events, such as creating a UUID during a model's `creating` event, you should call `Event::fake()` **after** using your factories.

Faking A Subset Of Events

If you only want to fake event listeners for a specific set of events, you may pass them to the `fake` or `fakeFor` method:

```
/**
```

```
     * Test order process.
     */
    public function testOrderProcess()
    {
        Event::fake([
            OrderCreated::class,
        ]);

        $order = factory(Order::class)->create();

        Event::assertDispatched(OrderCreated::class);

        // Other events are dispatched as normal...
        $order->update([...]);
    }
```

Scoped Event Fakes

If you only want to fake event listeners for a portion of your test, you may use the `fakeFor` method:

```
<?php

namespace Tests\Feature;

use App\Events\OrderCreated;
use App\Order;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Test order process.
     */
    public function testOrderProcess()
    {
        $order = Event::fakeFor(function () {
            $order = factory(Order::class)->create();

            Event::assertDispatched(OrderCreated::class);

            return $order;
        });

        // Events are dispatched as normal and observers will run ...
        $order->update([...]);
    }
}
```

Mail Fake

You may use the `Mail` facade's `fake` method to prevent mail from being sent. You may then assert that mailables were sent to users and even inspect the data they received. When using fakes, assertions are made after the code under test is executed:

```
<?php

namespace Tests\Feature;

use App\Mail\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Mail;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Mail::fake();

        // Assert that no mailables were sent...
        Mail::assertNothingSent();

        // Perform order shipping...

        Mail::assertSent(OrderShipped::class, function ($mail) use ($order) {
            return $mail->order->id === $order->id;
        });

        // Assert a message was sent to the given users...
        Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {
            return $mail->hasTo($user->email) &&
                   $mail->hasCc('...') &&
                   $mail->hasBcc('...');
        });

        // Assert a mailable was sent twice...
        Mail::assertSent(OrderShipped::class, 2);

        // Assert a mailable was not sent...
        Mail::assertNotSent(AnotherMailable::class);
    }
}
```

If you are queueing mailables for delivery in the background, you should use the `assertQueued` method instead of `assertSent`:

```
Mail::assertQueued(...);
Mail::assertNotQueued(...);
```

Notification Fake

You may use the `Notification` facade's `fake` method to prevent notifications from being sent. You may then assert that notifications were sent to users and even inspect the data they received. When using fakes, assertions are made after the code under test is executed:

```
<?php

namespace Tests\Feature;

use App\Notifications\OrderShipped;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Notifications\AnonymousNotifiable;
use Illuminate\Support\Facades\Notification;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Notification::fake();

        // Assert that no notifications were sent...
        Notification::assertNothingSent();

        // Perform order shipping...

        Notification::assertSentTo(
            $user,
            OrderShipped::class,
            function ($notification, $channels) use ($order) {
                return $notification->order->id === $order->id;
            }
        );

        // Assert a notification was sent to the given users...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );

        // Assert a notification was not sent...
        Notification::assertNotSentTo(
            [$user], AnotherNotification::class
        );

        // Assert a notification was sent via Notification::route() method...
        Notification::assertSentTo(
            new AnonymousNotifiable, OrderShipped::class
        );

        // Assert Notification::route() method sent notification to the correct us
        Notification::assertSentTo(
            new AnonymousNotifiable,
            OrderShipped::class,
            function ($notification, $channels, $notifiable) use ($user) {
                return $notifiable->routes['mail'] === $user->email;
            }
        );
    }
}
```

Queue Fake

As an alternative to mocking, you may use the `Queue` facade's `fake` method to prevent jobs from being queued. You may then assert that jobs were pushed to the queue and even inspect the data they received. When using fakes, assertions are made after the code under test is executed:

```php
<?php

namespace Tests\Feature;

use App\Jobs\ShipOrder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Queue;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Queue::fake();

        // Assert that no jobs were pushed...
        Queue::assertNothingPushed();

        // Perform order shipping...

        Queue::assertPushed(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // Assert a job was pushed to a given queue...
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // Assert a job was pushed twice...
        Queue::assertPushed(ShipOrder::class, 2);

        // Assert a job was not pushed...
        Queue::assertNotPushed(AnotherJob::class);

        // Assert a job was pushed with a specific chain...
        Queue::assertPushedWithChain(ShipOrder::class, [
            AnotherJob::class,
            FinalJob::class
        ]);
    }
}
```

Storage Fake

The `Storage` facade's `fake` method allows you to easily generate a fake disk that, combined with the file generation utilities of the `UploadedFile` class, greatly simplifies the testing of file uploads. For example:

```php
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function testAlbumUpload()
    {
        Storage::fake('photos');

        $response = $this->json('POST', '/photos', [
            UploadedFile::fake()->image('photo1.jpg'),
            UploadedFile::fake()->image('photo2.jpg')
        ]);

        // Assert one or more files were stored...
        Storage::disk('photos')->assertExists('photo1.jpg');
        Storage::disk('photos')->assertExists(['photo1.jpg', 'photo2.jpg']);

        // Assert one or more files were not stored...
        Storage::disk('photos')->assertMissing('missing.jpg');
        Storage::disk('photos')->assertMissing(['missing.jpg', 'non-existing.jpg']
    }
}
```

By default, the `fake` method will delete all files in its temporary directory. If you would like to keep these files, you may use the "persistentFake" method instead.

Facades

Unlike traditional static method calls, facades may be mocked. This provides a great advantage over traditional static methods and grants you the same testability you would have if you were using dependency injection. When testing, you may often want to mock a call to a Laravel facade in one of your controllers. For example, consider the following controller action:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

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's `get` method:

```php
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Support\Facades\Cache;
use Tests\TestCase;

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

        $response = $this->get('/users');

        // ...
    }
}
```

! You should not mock the `Request` facade. Instead, pass the input you desire into the HTTP helper methods such as `get` and `post` when running your test. Likewise, instead of mocking the `Config` facade, call the `Config::set` method in your tests.

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

Release Notes
Getting Started
Routing
Blade Templates
Authentication
Authorization
Artisan Console
Database
Eloquent ORM
Testing

Resources

Laracasts
Laravel News
Laracon
Laracon EU
Laracon AU
Jobs
Certification
Forums

Partners

Vehikl
Tighten Co.
Kirschbaum
Byte 5
64Robots
Cubet
DevSquad
Ideil
Cyber-Duck
ABOUT YOU
Become A Partner

Ecosystem

Vapor
Forge
Envoyer
Horizon
Lumen
Nova
Echo
Valet
Mix
Spark
Cashier
Homestead
Dusk
Passport
Scout
Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.