Course - Laravel Framework

Mô phỏng

Khi thử nghiệm các ứng dụng Laravel, bạn có thể muốn "mô phỏng" các khía cạnh nhất định của ứng dụng của mình để chúng không thực sự được thực thi trong một bài kiểm tra.

Tags: mocking, mo phong, laravel

Giới thiệu

Khi kiểm tra các ứng dụng Laravel, bạn có thể muốn "mô phỏng" các khía cạnh nào đó của ứng dụng của mình để chúng không thực sự được thực thi trong một bài kiểm tra nào đó. Ví dụ: khi kiểm tra một controller gửi một event, bạn có thể muốn bắt chước các chương trình theo dõi event để chúng không thực sự được thực thi trong quá trình kiểm tra. Điều này cho phép bạn chỉ kiểm tra phản hồi HTTP response của controller mà không cần lo lắng về việc thực thi chương trình theo dõi event vì chương trình theo dõi event có thể được kiểm tra trong trường hợp thử nghiệm của riêng họ.

Laravel cung cấp các phương pháp hữu ích để mô phỏng các event, công việc và các mặt khác bên ngoài. Những hàm trợ giúp này chủ yếu cung cấp một class tiện ích trên Mockery để bạn không phải thực hiện các lệnh gọi phương thức Mockery phức tạp theo cách thủ công.

Mô phỏng các đối tượng

Khi mô phỏng một đối tượng sẽ được đưa vào ứng dụng của bạn thông qua vùng chứa dịch vụ của Laravel, bạn sẽ cần phải liên kết đối tượng bị được mô phỏng của mình vào container dưới dạng ràng buộc `instance`. Điều này sẽ hướng dẫn container sử dụng đối tượng được mô phỏng của bạn thay vì xây dựng chính thức đối tượng:

```
use App\Service;
use Mockery;
use Mockery\MockInterface;

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

Để làm cho việc này thuận tiện hơn, bạn có thể sử dụng phương thức `mock` được cung cấp bởi class testcase cơ bản của Laravel. Ví dụ sau tương đương với ví dụ trên:

```php
use App\Service;
use Mockery\MockInterface;

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

Bạn có thể sử dụng phương thức `partMock` khi bạn chỉ cần mô phỏng một vài phương thức của một đối tượng. Các phương thức không bị giả lập sẽ được thực thi bình thường khi được gọi:

```php
use App\Service;
use Mockery\MockInterface;

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

Tương tự, nếu bạn muốn theo dõi một đối tượng, class testcase cơ sở của Laravel sẽ cung cấp phương thức `spy` như một gói bọc xung quanh phương thức `Mockery::spy`. Spies tương tự như mô phỏng; tuy nhiên, spy ghi lại bất kỳ tương tác nào giữa các spies và code đang được kiểm tra, cho phép bạn đưa ra xác nhận assertion sau khi code được thực thi:

```php
use App\Service;

$spy = $this->spy(Service::class);

// ...

$spy->shouldHaveReceived('process');
```

Mô phỏng facade

Không giống như các lệnh gọi phương thức tĩnh truyền thống, các mặt tiền (bao gồm cả mặt tiền thời gian thực) có thể bị mô phỏng. Điều này cung cấp một lợi thế lớn so với các

phương thức tĩnh truyền thống và cung cấp cho bạn khả năng kiểm tra tương tự như bạn sẽ có nếu bạn đang sử dụng phương pháp chèn thư viện lệ thuộc theo truyền thống. Khi chạy bài kiểm tra, bạn có thể thường muốn giả lập một cuộc gọi đến một facade Laravel xảy ra trong một trong các controller của bạn. Ví dụ: hãy xem xét hành động của controller sau:

```php
<?php

namespace App\Http\Controllers;
use Illuminate\Support\Facades\Cache;

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

    //
  }
}
```

Chúng ta có thể giả lập cuộc gọi đến facade `Cache` bằng cách sử dụng phương thức `shouldReceive`, phương thức này sẽ trả về một đối tượng của mô phỏng Mockery. Vì các facade thực sự được giải quyết và quản lý bởi service container Laravel, chúng có khả năng kiểm tra cao hơn nhiều so với một class tĩnh điển hình. Ví dụ, hãy giả lập cuộc gọi của chúng ta tới phương thức `get` của facade `Cache`:

```php
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
```

```php
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');

      // ...
  }
}
```

Chú ý: Bạn không nên giả lập facade Request. Thay vào đó, hãy truyền dữ liệu input mà bạn muốn vào các phương thức HTTP testing chẳng hạn như `get` và `post` khi chạy bài kiểm tra của bạn. Tương tự như vậy, thay vì mô phỏng facade `Config`, thì hãy gọi phương thức `Config::set` trong các bài kiểm tra của bạn.

Các facade spies

Nếu bạn muốn spy một facade, bạn có thể gọi phương thức `spy` trên facade tương xứng. Spies tương tự như mô phỏng; tuy nhiên, spies sẽ ghi lại bất kỳ tương tác nào giữa spies và code đang được kiểm tra, cho phép bạn đưa ra xác nhận assertion sau khi code được thực thi:

```php
use Illuminate\Support\Facades\Cache;

public function test_values_are_be_stored_in_cache()
{
  Cache::spy();

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

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

  Cache::shouldHaveReceived('put')->once()->with('name', 'Taylor', 10);
}
```

Bus Fake (giả lập)

Khi kiểm tra code phát tán công việc, bạn thường muốn xác nhận assertion rằng một công việc nào đó đã được gửi đi nhưng không thực sự xếp hàng chờ hoặc thực thi các tác vụ. Điều này là do việc thực thi tác vụ thường có thể được kiểm tra trong một class kiểm tra riêng biệt.

Bạn có thể sử dụng phương thức `fake` của facade `Bus` để ngăn các tác vụ được gửi đến hàng chờ. Sau đó, sau khi thực thi code đang được kiểm tra, bạn có thể kiểm tra tác vụ nào mà ứng dụng đã cố gắng phát tán đi bằng cách sử dụng các phương thức `assertDispatched` và `assertNotDispatched`:

```
<?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 test_orders_can_be_shipped()
  {
    Bus::fake();

    // Perform order shipping...

    // Assert that a job was dispatched...
```

```
        Bus::assertDispatched(ShipOrder::class);

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

        // Assert that a job was dispatched synchronously...
        Bus::assertDispatchedSync(AnotherJob::class);

        // Assert that a job was not dipatched synchronously...
        Bus::assertNotDispatchedSync(AnotherJob::class);

        // Assert that a job was dispatched after the response was sent...
        Bus::assertDispatchedAfterResponse(AnotherJob::class);

        // Assert a job was not dispatched after response was sent...
        Bus::assertNotDispatchedAfterResponse(AnotherJob::class);

        // Assert no jobs were dispatched...
        Bus::assertNothingDispatched();
    }
}
```

Bạn có thể truyền một hàm xử lý đối với các phương thức có sẵn để khẳng định rằng một tác vụ đã được cử đi đã vượt qua một "bài kiểm tra sự thật" nhất định. Nếu ít nhất một tác vụ được cử đi vượt qua bài kiểm tra sự thật đã cho thì việc xác nhận assertion sẽ thành công. Ví dụ: bạn có thể muốn xác nhận assertion rằng một tác vụ đã được điều động cho một đơn hàng cụ thể:

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

Xâu chuỗi các tác vụ

Phương thức `assertChained` của facade `Bus` có thể được sử dụng để khẳng định assertion rằng một chuỗi tác vụ đã được điều động. Phương thức `assertChained` chấp nhận một mảng các tác vụ được xâu chuỗi làm đối số đầu tiên của nó:

```php
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Bus;

Bus::assertChained([
  ShipOrder::class,
  RecordShipment::class,
  UpdateInventory::class
]);
```

Như bạn có thể thấy trong ví dụ trên, mảng các tác vụ được xâu chuỗi có thể là một mảng tên class của tác vụ. Tuy nhiên, bạn cũng có thể cung cấp một loạt các trường hợp tác vụ thực tế. Khi làm như vậy, Laravel sẽ đảm bảo rằng các đối tượng tác vụ thuộc cùng một class và có cùng giá trị thuộc tính của các công việc được xâu chuỗi mà ứng dụng của bạn gửi đi:

```php
Bus::assertChained([
  new ShipOrder,
  new RecordShipment,
  new UpdateInventory,
]);
```

Tác vụ tự động hóa

Phương thức `assertBatched` của facade `Bus` có thể được sử dụng để xác nhận rằng assertion một loạt tác vụ đã được gửi đi. Hàm xử lý được cung cấp cho phương thức `assertBatched` sẽ nhận được một thể hiện của `Illuminate\Bus\PendingBatch`, có thể được sử dụng để kiểm tra các công việc trong lô:

```php
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;

Bus::assertBatched(function (PendingBatch $batch) {
  return $batch->name == 'import-csv' && $batch->jobs->count() === 10;
});
```

Giả lập event

Khi kiểm tra code phát tán event, bạn có thể muốn hướng dẫn Laravel không thực sự thực thi chương trình theo dõi event. Khi sử dụng phương thức `fake` của facade `Event`, bạn có thể chặn chương trình theo dõi khỏi việc thực thi, thực thi code đang được kiểm tra và sau đó xác nhận assertion các sự kiện nào đã được ứng dụng của bạn phát tán đi bằng các phương thức `assertDispatched`, `assertNotDispatched`, và `assertNothingDispatched`:

```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 test_orders_can_be_shipped()
  {
    Event::fake();
```

```
        // Perform order shipping...

        // Assert that an event was dispatched...
        Event::assertDispatched(OrderShipped::class);

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

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

        // Assert that no events were dispatched...
        Event::assertNothingDispatched();
    }
}
```

Bạn có thể truyền một hàm xử lý đối với các phương thức `assertDispatched` hoặc `assertNotDispatched` để khẳng định rằng một event đã được gửi đi thực sự đã vượt qua một "bài kiểm tra sự thật" nhất định. Nếu ít nhất một event được gửi đi vượt qua bài kiểm tra sự thật đã cho thì xác nhận assertion sẽ thành công:

```
Event::assertDispatched(function (OrderShipped $event) use ($order) {
    return $event->order->id === $order->id;
});
```

Nếu bạn chỉ muốn khẳng định rằng một chương trình theo dõi event đang theo dõi một event nhất định, bạn có thể sử dụng phương thức `assertListening`:

```
Event::assertListening(
    OrderShipped::class,
    SendShipmentNotification::class
);
```

> **Chú ý:** Sau khi gọi `Event::fake()`, không có chương trình xử lý event nào được thực thi. Vì vậy, nếu các bài kiểm tra của bạn sử dụng các factory model dựa vào các event, chẳng hạn như tạo UUID trong event `creating` của model, bạn nên gọi

`Event::fake()` sau khi sử dụng các factory của mình.

Giả lập tập hợp các Event

Nếu bạn chỉ muốn giả lập chương trình theo dõi event cho một nhóm event cụ thể, bạn có thể truyền chúng vào phương thức `fake` hoặc `fakeFor`.

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

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

  Event::assertDispatched(OrderCreated::class);

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

Định vị các fake của event

Nếu bạn chỉ muốn giả mạo người nghe sự kiện cho một phần thử nghiệm của mình, bạn có thể sử dụng phương pháp `fakeFor`:

```
<?php

namespace Tests\Feature;

use App\Events\OrderCreated;
use App\Models\Order;
use Illuminate\Foundation\Testing\RefreshDatabase;
```

```php
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

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

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

Giả lập HTTP

Phương thức fake của facade Http cho phép bạn hướng dẫn máy khách HTTP client trả về phản hồi stubbed/dummy (giả/nhái) khi yêu cầu được thực hiện. Để biết thêm thông tin về việc giả lập các yêu cầu HTTP gửi đi, vui lòng tham khảo tài liệu kiểm tra Ứng dụng khách HTTP client.

Giả lập Mail

Bạn có thể sử dụng phương thức fake của facade Mail để chặn thư được phán tái đi. Thông thường, việc gửi thư không liên quan đến code bạn đang thực sự thử nghiệm. Rất có thể, chỉ cần khẳng định rằng Laravel đã được hướng dẫn để gửi một mailable nào đó.

Sau khi gọi phương thức fake của facade Mail, bạn có thể khẳng định rằng mailables đã được hướng dẫn để gửi đến người dùng và thậm chí kiểm tra dữ liệu mà mailables nhận

được:

```php
<?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 test_orders_can_be_shipped()
  {
    Mail::fake();

    // Perform order shipping...

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

    // Assert that a mailable was sent...
    Mail::assertSent(OrderShipped::class);

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

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

Nếu bạn đang xếp hàng chờ các hộp thư để gửi đi trong nền, bạn nên sử dụng phương thức `assertQueued` thay vì `assertSent`:

```php
Mail::assertQueued(OrderShipped::class);

Mail::assertNotQueued(OrderShipped::class);

Mail::assertNothingQueued();
```

Bạn có thể truyền một hàm xử lý đối với các phương thức `assertSent`, `assertNotSent`, `assertQueued` hoặc `assertNotQueued` sẽ xác nhận rằng một thư mailable đã gửi đã vượt qua một "bài kiểm tra sự thật" nhất định. Nếu ít nhất một thư mailable được gửi đã vượt qua bài kiểm tra sự thật đã cho thì xác nhận assertion sẽ thành công:

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

Khi gọi các phương thức xác nhận assertion của facade Mail, đối tượng mailable được chấp nhận bởi hàm xử lý được cung cấp sẽ tiết lộ các phương thức hữu ích để kiểm tra người nhận của thư mailable:

```php
Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {
  return $mail->hasTo($user->email) &&
    $mail->hasCc('...') &&
    $mail->hasBcc('...');
});
```

Bạn có thể nhận thấy rằng có hai phương thức để xác nhận rằng thư không được gửi đi: `assertNotSent` và `assertNotQueued`. Đôi khi bạn có thể muốn khẳng định rằng không có thư nào được gửi hoặc xếp hàng đợi. Để thực hiện điều này, bạn có thể sử dụng các phương thức `assertNothingOutgoing` và `assertNotOutgoing`:

```php
Mail::assertNothingOutgoing();

Mail::assertNotOutgoing(function (OrderShipped $mail) use ($order) {
  return $mail->order->id === $order->id;
});
```

Giả lập thông báo

Bạn có thể sử dụng phương thức `fake` của facade `Notification` để chặn thông báo được gửi đi. Thông thường, việc gửi thông báo không liên quan đến code bạn đang thực sự kiểm định. Rất có thể, chỉ cần khẳng định rằng Laravel đã được hướng dẫn để gửi một thông báo nhất định.

Sau khi gọi phương thức `fake` của facade `Notification`, bạn có thể khẳng định rằng thông báo đã được hướng dẫn để gửi tới người dùng và thậm chí thanh tra dữ liệu mà thông báo nhận được:

```php
<?php

namespace Tests\Feature;

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

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

    // Perform order shipping...

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

    // 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
    );
  }
}
```

Bạn có thể truyền một hàm xử lý đối với các phương thức `assertSentTo` hoặc `assertNotSentTo` để xác nhận rằng thông báo đã được gửi vượt qua một "bài kiểm tra sự thật" nhất định. Nếu ít nhất một thông báo được gửi vượt qua bài kiểm tra sự thật nhất định thì xác nhận sẽ thành công:

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

Thông báo on-demand

Nếu code bạn đang kiểm tra gửi thông báo theo yêu cầu, thì bạn sẽ cần xác nhận rằng thông báo đã được gửi đến đối tượng `Illuminate\Notifications\AnonymousNotifiable`:

```
use Illuminate\Notifications\AnonymousNotifiable;

Notification::assertSentTo(
  new AnonymousNotifiable, OrderShipped::class
);
```

Bằng việc truyền một hàm xử lý làm đối số thứ ba cho các phương thức xác nhận thông báo, bạn có thể xác định xem thông báo theo yêu cầu có được gửi đến đúng địa chỉ "route" hay không:

```
Notification::assertSentTo(
  new AnonymousNotifiable,
```

```
    OrderShipped::class,
    function ($notification, $channels, $notifiable) use ($user) {
        return $notifiable->routes['mail'] === $user->email;
    }
);
```

Giả lập Queue

Bạn có thể sử dụng phương thức fake của facade Queue để ngăn các tác vụ đã xếp hàng được đẩy vào hàng đợi. Rất có thể, chỉ cần khẳng định rằng Laravel đã được hướng dẫn để đẩy một tác vụ đã cho vào hàng đợi vì bản thân các tác vụ đã xếp hàng có thể được kiểm tra trong một class thử nghiệm khác.

Sau khi gọi phương thức fake của facade Queue, bạn có thể khẳng định rằng ứng dụng đã cố gắng đẩy tác vụ vào hàng đợi:

```
<?php

namespace Tests\Feature;

use App\Jobs\AnotherJob;
use App\Jobs\FinalJob;
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 test_orders_can_be_shipped()
    {
        Queue::fake();

        // Perform order shipping...

        // Assert that no jobs were pushed...
```

```
        Queue::assertNothingPushed();

        // 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);
    }
}
```

Bạn có thể truyền một hàm xử lý đối với các phương thức `assertPushed` hoặc `assertNotPush` để xác nhận rằng một tác vụ đã được đẩy vào đã vượt qua một "bài kiểm tra sự thật" nhất định. Nếu ít nhất một tác vụ được thúc đẩy vượt qua bài kiểm tra sự thật đã cho thì khẳng định assertion sẽ thành công:

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

Xâu chuỗi các tác vụ

Có thể sử dụng các phương thức `assertPushedWithChain` và `assertPushedWithChain` của facade `Queue` để kiểm tra chuỗi tác vụ của một tác vụ đã được đẩy vào. Phương thức `assertPushedWithChain` chấp nhận tác vụ chính làm đối số đầu tiên của nó và một mảng các tác vụ được xâu chuỗi làm đối số thứ hai của nó:

```
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Queue;

Queue::assertPushedWithChain(ShipOrder::class, [
    RecordShipment::class,
```

```
    UpdateInventory::class
]);
```

Như bạn có thể thấy trong ví dụ trên, mảng các tác vụ được xâu chuỗi có thể là một mảng tên class của tác vụ. Tuy nhiên, bạn cũng có thể cung cấp một loạt các đối tượng tác vụ thực tế. Khi làm như vậy, Laravel sẽ đảm bảo rằng các đối tượng công việc thuộc cùng một class và có cùng giá trị thuộc tính của các tác vụ được xâu chuỗi mà ứng dụng của bạn gửi đi:

```
Queue::assertPushedWithChain(ShipOrder::class, [
    new RecordShipment,
    new UpdateInventory,
]);
```

Bạn có thể sử dụng phương thức `assertPushedWithoutChain` để khẳng định rằng công việc đã được đẩy mà không có một chuỗi tác vụ nào:

```
Queue::assertPushedWithoutChain(ShipOrder::class);
```

Giả lập trong lưu trữ tập tin

Phương thức `fake` của facade `Storage` cho phép bạn dễ dàng tạo một đĩa giả, kết hợp với các tiện ích tạo tập tin của class `Illuminate\Http\UploadedFile`, giúp đơn giản hóa đáng kể việc kiểm tra tải lên tập tin. Ví dụ:

```
<?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 test_albums_can_be_uploaded()
  {
    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']);
  }
}
```

Để biết thêm thông tin về kiểm tra tải lên tập tin, bạn có thể tham khảo thông tin của tài liệu kiểm tra HTTP về upload lên tập tin.

> **Chú ý:** Theo mặc định, phương thức `fake` sẽ xóa tất cả các tập tin trong thư mục tạm thời của nó. Nếu bạn muốn giữ lại các tập tin này, bạn có thể sử dụng phương thức "secureFake" để thay thế.

Tương tác với thời gian

Khi kiểm tra, đôi khi bạn có thể cần phải sửa đổi thời gian do hàm trợ giúp trả về, chẳng hạn như `now` hoặc `Illuminate\Support\Carbon::now()`. Rất may, class kiểm tra tính năng cơ sở của Laravel bao gồm các hàm trợ giúp cho phép bạn thao tác với thời gian hiện tại:

```
public function testTimeCanBeManipulated()
{
  // Travel into the future...
  $this->travel(5)->milliseconds();
```

```php
    $this->travel(5)->seconds();
    $this->travel(5)->minutes();
    $this->travel(5)->hours();
    $this->travel(5)->days();
    $this->travel(5)->weeks();
    $this->travel(5)->years();

    // Travel into the past...
    $this->travel(-5)->hours();

    // Travel to an explicit time...
    $this->travelTo(now()->subHours(6));

    // Return back to the present time...
    $this->travelBack();
}
```