Course - Laravel Framework

Testing

Laravel được xây dựng với mục đích kiểm định. Trên thực tế, việc kiểm định được hỗ trợ với PHPUnit đã được đưa vào sẵn và tập tin phpunit.xml đã được thiết lập cho ứng dụng của bạn. Framework này cũng đi kèm với các phương thức tiện ích cho phép bạn kiểm định các ứng dụng của mình một cách rõ ràng.

Tags: testing, kiem nghiem, laravel

Giới thiệu

Laravel được xây dựng với mục đích kiểm định. Trên thực tế, việc kiểm định được hỗ trợ với PHPUnit đã được đưa vào sẵn và tập tin *phpunit.xml* đã được thiết lập cho ứng dụng của bạn. Framework này cũng đi kèm với các phương thức tiện ích cho phép bạn kiểm định các ứng dụng của mình một cách rõ ràng.

Theo mặc định, thư mục *tests* ứng dụng của bạn chứa hai thư mục: *Feature* và *Unit*. Bài kiểm tra Unit (đơn vị lẻ) là bài kiểm tra tập trung vào một phần rất nhỏ, tách biệt trong mã của bạn. Trên thực tế, hầu hết các bài kiểm tra unit tập trung vào một phương thức đơn lẻ nào đó. Các bài kiểm tra trong thư mục *Unit* của bạn không khởi động ứng dụng Laravel của bạn và do đó không thể truy cập cơ sở dữ liệu của ứng dụng hoặc các dịch vụ khác của framework.

Kiểm tra tính năng có thể sẽ kiểm tra một số lượng lớn code của bạn, bao gồm cách một số đối tượng tương tác với nhau hoặc thậm chí là một yêu cầu HTTP request đầy đủ tới điểm cuối JSON. Nói chung, hầu hết các bài kiểm tra của bạn phải là các bài kiểm tra tính năng. Những loại bài kiểm tra này cung cấp sự tin tưởng nhất định rằng toàn bộ hệ thống của bạn đang hoạt động như dự kiến.

Tập tin *ExampleTest.php* được cung cấp trong cả thư mục kiểm định *Feature* và *Unit*. Sau khi thiết lập ứng dụng Laravel mới, hãy thực thi các lệnh kiểm định `vendor/bin/phpunit` hay `php artisan test` để chạy các bài kiểm tra của bạn.

Môi trường

Khi chạy bài kiểm tra, Laravel sẽ tự động thiết lập môi trường cấu hình cho `testing` vì các biến môi trường được xác định trong tập tin *phpunit.xml*. Laravel cũng tự động cấu hình session và bộ nhớ cache cho driver array trong khi kiểm định, có nghĩa là không có dữ liệu session hoặc bộ nhớ cache nào được duy trì trong khi kiểm tra.

Bạn có thể tự do khai báo các giá trị cấu hình môi trường kiểm định khác nếu cần. Các biến môi trường kiểm định có thể được cấu hình trong tập tin *phpunit.xml* của ứng dụng của bạn, nhưng hãy đảm bảo là đã xóa bộ nhớ cache cấu hình của bạn bằng cách sử dụng lệnh Artisan `config:clear` trước khi chạy các bài kiểm tra của bạn!

Tập tin môi trường *.env.testing*

Ngoài ra, bạn có thể tạo tập tin *.env.testing* trong thư mục gốc (root) của dự án. Tập tin này sẽ được sử dụng thay cho tập tin *.env* khi chạy các bài kiểm tra PHPUnit hoặc thực hiện các lệnh Artisan với tùy chọn `--env=testing`.

Trait `CreatesApplication`

Laravel đưa vào trait `CreatesApplication` được áp dụng cho class `TestCase` cơ sở của ứng dụng của bạn. Trait này chứa một phương thức `createApplication` sẽ khởi động ứng dụng Laravel trước khi chạy các bài kiểm tra của bạn. Điều quan trọng là bạn phải để trait này ở vị trí ban đầu của nó vì một số tính năng, chẳng hạn như tính năng kiểm tra song song của Laravel, phụ thuộc vào nó.

Tạo bài kiểm định

Để tạo một testcase mới, hãy sử dụng lệnh Artisan `make:test`. Theo mặc định, các bài kiểm tra sẽ được đặt trong thư mục *tests/Feature*:

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

Nếu bạn muốn tạo một bài kiểm tra trong thư mục *tests/Unit*, bạn có thể sử dụng tuỳ chọn `--unit` khi thực hiện lệnh `make:test`:

```
php artisan make:test UserTest --unit
```

Nếu bạn muốn tạo một bài kiểm tra PHP Pest, bạn có thể cung cấp tùy chọn `--pest` cho lệnh `make:test`:

```
php artisan make:test UserTest --pest
```

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

Bài kiểm tra stub có thể được tùy chỉnh bằng cách sử dụng stub publishing.

Khi bài kiểm tra đã được tạo, bạn có thể khai báo các phương thức kiểm tra như bạn thường làm khi sử dụng PHPUnit. Để chạy các bài kiểm tra của bạn, hãy thực hiện lệnh `vendor/bin/phpunit` hoặc `php artisan test` trên cửa sổ lệnh terminal của bạn:

```php
<?php

namespace Tests\Unit;
use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
```

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

Chú ý: Nếu bạn khai báo các phương thức `setUp`/`setDown` của riêng mình trong một class kiểm định, hãy đảm bảo gọi các phương thức `parent::setUp()`/`parent::setDown()` tương ứng trên class mẹ.

Chạy các bài kiểm tra

Như đã đề cập trước đây, khi bạn đã viết các bài kiểm tra, bạn có thể chạy chúng bằng `phpunit`

```
./vendor/bin/phpunit
```

Ngoài lệnh `phpunit`, bạn có thể sử dụng lệnh Artisan `test` để chạy các bài kiểm tra của mình. Khi chạy bài kiểm tra Artisan cung cấp các dòng báo cáo phản hồi liên tục để dễ dàng phát triển và gỡ lỗi:

```
php artisan test
```

Bất kỳ đối số nào có thể được truyền cho lệnh phpunit cũng có thể được truyền cho lệnh Artisan `test`:

```
php artisan test --testsuite=Feature --stop-on-failure
```

Chạy song song các bài kiểm tra

Theo mặc định, Laravel và PHPUnit thực hiện có trật tự các bài kiểm tra của bạn theo một

quy trình. Tuy nhiên, bạn có thể giảm đáng kể thời gian chạy các bài kiểm tra của mình bằng cách chạy đồng thời trên nhiều quy trình. Để bắt đầu, hãy đảm bảo ứng dụng của bạn dựa trên phiên bản `^5.3` trở lên của package `nunomaduro/collision`. Sau đó, bao gồm tùy chọn `--parallel` khi thực hiện lệnh Artisan `test`:

```
php artisan test --parallel
```

Theo mặc định, Laravel sẽ tạo càng nhiều quy trình nhất có thể tùy vào khả năng của CPU trên máy tính của bạn. Tuy nhiên, bạn có thể điều chỉnh số lượng quy trình bằng cách sử dụng tùy chọn `--processes`:

```
php artisan test --parallel --processes=4
```

> **Chú ý:** Khi chạy song song các bài kiểm tra, một số tùy chọn PHPUnit (chẳng hạn như `--do-not-cache-result`) có thể không khả dụng.

Kiểm định song song và cơ sở dữ liệu

Laravel tự động xử lý việc tạo và migrate cơ sở dữ liệu thử nghiệm cho mỗi quy trình xử lý song song đang chạy thử nghiệm của bạn. Các cơ sở dữ liệu thử nghiệm sẽ được gắn với một mã token quy trình thống nhất cho mỗi quá trình. Ví dụ: nếu bạn có hai quy trình kiểm tra song song, Laravel sẽ tạo và sử dụng cơ sở dữ liệu kiểm tra `your_db_test_1` và `your_db_test_2`.

Theo mặc định, cơ sở dữ liệu thử nghiệm vẫn tồn tại giữa các lần gọi đến lệnh Artisan `test` để chúng có thể được sử dụng lại bằng các lần gọi `test` tiếp theo. Tuy nhiên, bạn có thể tạo lại chúng bằng cách sử dụng tùy chọn `--recreate-databases`:

```
php artisan test --parallel --recreate-databases
```

Kẹp các bài kiểm tra song song

Đôi khi, bạn có thể cần chuẩn bị một số tài nguyên nhất định được sử dụng bởi các bài kiểm tra của ứng dụng của bạn để chúng có thể được sử dụng một cách an toàn bởi nhiều quy trình thử nghiệm.

Sử dụng facade `ParallelTesting`, bạn có thể chỉ định mã sẽ được thực thi trên `setUp` và `tearDown` của một quy trình hoặc testcase. Các hàm xử lý đã cho sẽ nhận được các biến `$token` và `$testCase` có chứa mã token của quy trình và testcase hiện tại, tương ứng:

```php
<?php

namespace App\Providers;
use Illuminate\Support\Facades\Artisan;
use Illuminate\Support\Facades\ParallelTesting;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
  /**
   * Bootstrap any application services.
   *
   * @return void
   */
  public function boot()
  {
    ParallelTesting::setUpProcess(function ($token) {
      // ...
    });

    ParallelTesting::setUpTestCase(function ($token, $testCase) {
      // ...
    });

    // Executed when a test database is created...
    ParallelTesting::setUpTestDatabase(function ($database, $token) {
      Artisan::call('db:seed');
    });

    ParallelTesting::tearDownTestCase(function ($token, $testCase) {
      // ...
    });

    ParallelTesting::tearDownProcess(function ($token) {
      // ...
    });
  }
```

```
}
```

Truy cập token kiểm định song song

Nếu bạn muốn truy cập mã "token" của các quy trình song song hiện tại từ bất kỳ vị trí nào khác trong code thử nghiệm của ứng dụng, thì bạn có thể sử dụng phương thức `token`. Mã token này là số nhận dạng kiểu chuỗi, và thống nhất cho một quy trình thử nghiệm riêng lẻ và có thể được sử dụng để phân đoạn tài nguyên qua các quy trình thử nghiệm song song. Ví dụ: Laravel sẽ tự động gắn mã token này vào cuối cơ sở dữ liệu thử nghiệm được tạo bởi mỗi quy trình thử nghiệm song song:

```
$token = ParallelTesting::token();
```

Báo cáo phạm trù bài kiểm tra

> **Chú ý:** Tính năng này yêu cầu *Xdebug* hoặc *PCOV*.

Khi chạy các bài kiểm tra ứng dụng, bạn có thể muốn xác định xem các testcase của mình có thực sự kiểm tra code của ứng dụng hay không và lượng code của ứng dụng được sử dụng khi chạy các bài kiểm tra. Để thực hiện điều này, bạn có thể thêm tùy chọn `--coverage` khi gọi lệnh `test`:

```
php artisan test --coverage
```

Kiểm định trong một phạm trù tối thiểu

Bạn có thể sử dụng tùy chọn `--min` để khai báo ngưỡng phạm trù kiểm tra tối thiểu cho ứng dụng của mình. Bộ thử nghiệm sẽ không thành công nếu ngưỡng này không được đáp ứng:

```
php artisan test --coverage --min=80.3
```