Course - Laravel Framework

Testing và database

Laravel cung cấp nhiều công cụ và xác nhận hữu ích để giúp bạn kiểm tra các ứng dụng điều khiển cơ sở dữ liệu dễ dàng hơn. Ngoài ra, các nhà máy và trình gieo mô hình Laravel giúp bạn dễ dàng tạo các bản ghi cơ sở dữ liệu thử nghiệm bằng cách sử dụng các mô hình và mối quan hệ Eloquent của ứng dụng của bạn. Chúng tôi sẽ thảo luận về tất cả các tính năng mạnh mẽ này trong tài liệu sau.

Tags: testing, testing va database, laravel

Giới thiệu

Laravel cung cấp nhiều công cụ và xác nhận hữu ích để giúp bạn kiểm tra các ứng dụng điều khiển cơ sở dữ liệu dễ dàng hơn. Ngoài ra, các nhà máy và trình gieo mô hình Laravel giúp bạn dễ dàng tạo các bản ghi cơ sở dữ liệu thử nghiệm bằng cách sử dụng các mô hình và mối quan hệ Eloquent của ứng dụng của bạn. Chúng tôi sẽ thảo luận về tất cả các tính năng mạnh mẽ này trong tài liệu sau.

Khôi phục lại cơ sở dữ liệu sau mỗi lần kiểm tra

Trước khi tiếp tục nhiều hơn nữa, hãy thảo luận cách khôi phục lại cơ sở dữ liệu của bạn sau mỗi lần kiểm tra để dữ liệu từ lần kiểm tra trước không ảnh hưởng đến các lần kiểm tra tiếp theo. Trait `Illuminate\Foundation\Testing\RefreshDatabase` có trong Laravel sẽ giải quyết vấn đề này cho bạn. Chỉ cần sử dụng trait trên class thử nghiệm của bạn:

```php
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
  use RefreshDatabase;

  /**
   * A basic functional test example.
   *
   * @return void
   */
  public function test_basic_example()
  {
    $response = $this->get('/');
    // ...
  }
```

```
}
```

Khai báo factory cho model

Khái niệm tổng quát

Đầu tiên, hãy nói về các factory cho Eloquent model. Khi kiểm tra, bạn có thể cần phải chèn một vài record vào cơ sở dữ liệu của mình trước khi thực hiện bài kiểm tra. Thay vì khai báo thủ công giá trị của từng cột khi bạn tạo dữ liệu kiểm tra này, Laravel cho phép bạn xác định một tập hợp các thuộc tính mặc định cho từng Eloquent model của bạn bằng cách sử dụng các factory model.

Để xem ví dụ về cách viết một factory, hãy xem tập tin *database/factories/UserFactory.php* trong ứng dụng của bạn. Factory này được bao gồm với tất cả các ứng dụng Laravel mới và chứa phần tạo lập factory như sau:

```
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
  /**
   * Define the model's default state.
   *
   * @return array
   */
  public function definition()
  {
    return [
      'name' => $this->faker->name(),
      'email' => $this->faker->unique()->safeEmail(),
      'email_verified_at' => now(),
      'password' => '$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
      'remember_token' => Str::random(10),
    ];
```

```
    }
}
```

Như bạn có thể thấy, ở dạng cơ bản nhất của chúng, factories là các class mở rộng class nhà máy cơ sở của Laravel và khai báo phương thức `definition`. Phương thức `definition` sẽ trả về tập hợp giá trị thuộc tính attribute mặc định sẽ được áp dụng khi tạo model bằng cách sử dụng factory.

Thông qua thuộc tính `faker`, các nhà máy có quyền truy cập vào thư viện Faker PHP, cho phép bạn tạo các loại dữ liệu ngẫu nhiên khác nhau một cách thuận tiện để kiểm tra.

> Bạn có thể đặt ngôn ngữ Faker cho ứng dụng của mình bằng cách thêm tùy chọn `faker_locale` vào tập tin cấu hình *config/app.php* của mình.

Tạo ra các factory

Để tạo một factory, hãy thực hiện lệnh Artisan `make:factory`:

```
php artisan make:factory PostFactory
```

Class factory mới sẽ được đặt trong thư mục *database/factories* của bạn.

Quy ước Công khai model & factory

Khi bạn đã tạo các factory của mình, bạn có thể sử dụng phương thức tĩnh `factory` được cung cấp cho các model của bạn bởi trait `Illuminate\Database\Eloquent\Factories\HasFactory` để khởi tạo một đối tượng factory cho model đó.

Phương pháp `factory` của trait `HasFactory` sẽ sử dụng các quy ước để xác định nhà máy phù hợp cho model mà trait được chỉ định. Cụ thể, phương thức sẽ tìm kiếm một nhà máy trong namespace `Database\Factories` có tên class khớp với tên model và có hậu tố là `Factory`. Nếu các quy ước này không áp dụng cho ứng dụng hoặc factory cụ thể của bạn, bạn có thể ghi đè phương thức `newFactory` trên model của mình để trả về trực tiếp một đối tượng của nhà máy tương ứng của model:

```
use Database\Factories\Administration\FlightFactory;

/**
 * Create a new factory instance for the model.
```

```
     *
     * @return \Illuminate\Database\Eloquent\Factories\Factory
     */
    protected static function newFactory()
    {
        return FlightFactory::new();
    }
```

Tiếp theo, xác định thuộc tính `model` trên nhà máy tương ứng:

```
use App\Administration\Flight;
use Illuminate\Database\Eloquent\Factories\Factory;

class FlightFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     *
     * @var string
     */
    protected $model = Flight::class;
}
```

Các loại factory

Các phương thức thao tác trạng thái cho phép bạn xác định các sửa đổi rời rạc có thể được áp dụng cho các factory của model của bạn trong bất kỳ sự kết hợp nào. Ví dụ: factory `Database\Factories\UserFactory` của bạn có thể chứa một phương thức trạng thái `suspended` để sửa đổi một trong các giá trị thuộc tính attribute mặc định của nó.

Các phương thức chuyển đổi trạng thái thường gọi phương thức `state` được cung cấp bởi class factory cơ sở của Laravel. Phương thức `state` chấp nhận một hàm xử lý sẽ nhận mảng thuộc tính attribute thô được khai báo cho factory và sẽ trả về một mảng thuộc tính attribute để sửa đổi:

```
/**
 * Indicate that the user is suspended.
```

```
     *
     * @return \Illuminate\Database\Eloquent\Factories\Factory
     */
    public function suspended()
    {
        return $this->state(function (array $attributes) {
            return [
                'account_status' => 'suspended',
            ];
        });
    }
```

Lệnh Callback Factory

Các lệnh gọi lại ban đầu được đăng ký bằng cách sử dụng các phương thức afterMaking và afterCreating và cho phép bạn thực hiện các tác vụ bổ sung sau khi tạo một model. Bạn nên đăng ký các lệnh gọi lại này bằng cách khai báo phương thức `configure` trên lớp factory của bạn. Phương thức này sẽ được Laravel tự động gọi khi khởi tạo factory:

```
namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    /**
     * Configure the model factory.
     *
     * @return $this
     */
    public function configure()
    {
        return $this->afterMaking(function (User $user) {
            //
        })->afterCreating(function (User $user) {
```

```
        //
    });
  }

  // ...
}
```

Tạo model bằng các factory

Tạo model

Khi bạn đã tạo các factory của mình, bạn có thể sử dụng phương thức tĩnh `factory` được cung cấp cho các model của bạn bởi trait `Illuminate\Database\Eloquent\Factories\HasFactory` để khởi tạo một đối tượng factory cho model đó. Chúng ta hãy xem một vài ví dụ về việc tạo model. Đầu tiên, chúng ta sẽ sử dụng phương thức `make` để tạo các model mà không cần lưu chúng vào cơ sở dữ liệu:

```
use App\Models\User;

public function test_models_can_be_instantiated()
{
  $user = User::factory()->make();

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

Bạn có thể tạo một collection nhiều model bằng cách sử dụng phương pháp `count`:

```
$users = User::factory()->count(3)->make();
```

Áp dụng các States

Bạn cũng có thể áp dụng bất kỳ trạng thái nào của mình cho các model. Nếu bạn muốn áp dụng nhiều phép biến đổi trạng thái cho các model, thì bạn có thể chỉ cần gọi trực tiếp các phương thức chuyển đổi trạng thái:

```php
$users = User::factory()->count(5)->suspended()->make();
```

Ghi chồng các thuộc tính attributes

Nếu bạn muốn ghi đè một số giá trị mặc định của các model của mình, bạn có thể truyền một mảng giá trị vào phương thức `make`. Chỉ các thuộc tính attribute được mô tả sẽ được thay thế trong khi phần còn lại của các thuộc tính attribute vẫn được đặt thành giá trị mặc định của chúng như được mô tả bởi factory:

```php
$user = User::factory()->make([
  'name' => 'Abigail Otwell',
]);
```

Ngoài ra, phương thức state có thể được gọi trực tiếp trên đối tượng factory để thực hiện chuyển đổi trạng thái nội tuyến:

```php
$user = User::factory()->state([
  'name' => 'Abigail Otwell',
])->make();
```

> Tính năng *bảo vệ phân công* (Mass assignment protection) sẽ tự động bị vô hiệu hóa khi tạo model bằng cách sử dụng factory.

Cập nhật các model

Phương thức `create` khởi tạo các đối tượng model và lưu trữ chúng vào cơ sở dữ liệu bằng cách sử dụng phương thức `save` của Eloquent:

```php
use App\Models\User;

public function test_models_can_be_persisted()
{
  // Create a single App\Models\User instance...
  $user = User::factory()->create();
```

```
    // Create three App\Models\User instances...
    $users = User::factory()->count(3)->create();

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

Bạn có thể ghi đè các thuộc tính attribute model mặc định của factory bằng cách truyền một mảng thuộc tính attribute vào phương thức `create`:

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

Trình tự

Đôi khi bạn có thể muốn thay thế giá trị của một thuộc tính attribute model nhất định cho mỗi model đã tạo. Bạn có thể thực hiện điều này bằng cách khai báo một sự chuyển đổi trạng thái dưới dạng một xâu chuỗi. Ví dụ: bạn có thể muốn thay thế giá trị của cột `admin` giữa `Y` và `N` cho mỗi người dùng đã tạo:

```
use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Sequence;

$users = User::factory()
    ->count(10)
    ->state(new Sequence(
        ['admin' => 'Y'],
        ['admin' => 'N'],
    ))
    ->create();
```

Trong ví dụ này, năm người dùng sẽ được tạo với giá trị `admin` là `Y` và năm người dùng sẽ được tạo với giá trị `admin` là `N`.

Nếu cần, bạn có thể bao gồm một hàm xử lý làm giá trị trình tự. Hàm xử lý sẽ được gọi mỗi khi chuỗi cần một giá trị mới:

```php
$users = User::factory()
  ->count(10)
  ->state(new Sequence(
    fn ($sequence) => ['role' => UserRoles::all()->random()],
  ))
  ->create();
```

Trong một hàm xử lý trình tự, bạn có thể truy cập thuộc tính `$index` hoặc `$count` trên đối tượng xâu chuỗi được đưa vào hàm xử lý. Thuộc tính `$index` chứa số lần lặp qua chuỗi đã xảy ra cho đến nay, trong khi thuộc tính `$count` chứa tổng số lần chuỗi sẽ được gọi:

```php
$users = User::factory()
  ->count(10)
  ->sequence(fn ($sequence) => ['name' => 'Name '.$sequence->index])
  ->create();
```

Mối quan hệ của factory

Mối quan hệ has Many

Tiếp theo, chúng ta hãy khám phá việc xây dựng các mối quan hệ theo Eloquent model bằng cách sử dụng các phương thức factory thông thạo của Laravel. Trước tiên, hãy giả sử ứng dụng của chúng ta có model `App\Models\User` và một model `App\Models\Post`. Ngoài ra, hãy giả sử rằng model User khai báo mối quan hệ `hasMany` với `Post`. Chúng ta có thể tạo một người dùng có ba bài đăng bằng phương thức `has` do các nhà máy của Laravel cung cấp. Phương thức `has` chấp nhận một đối tượng gốc:

```php
use App\Models\Post;
use App\Models\User;

$user = User::factory()
  ->has(Post::factory()->count(3))
  ->create();
```

Theo quy ước, khi truyền một model `Post` cho phương thức `has`, Laravel sẽ giả định rằng

model `User` phải có một phương thức `posts` xác định mối quan hệ. Nếu cần, bạn có thể chỉ định rõ ràng tên của mối quan hệ mà bạn muốn thao tác:

```
$user = User::factory()
  ->has(Post::factory()->count(3), 'posts')
  ->create();
```

Tất nhiên, bạn có thể thực hiện các thao tác trạng thái trên các model liên quan. Ngoài ra, bạn có thể truyền một hàm xử lý chuyển đổi trạng thái nếu sự thay đổi trạng thái của bạn yêu cầu quyền truy cập vào model mẹ:

```
$user = User::factory()
  ->has(
    Post::factory()
      ->count(3)
      ->state(function (array $attributes, User $user) {
        return ['user_type' => $user->type];
      })
  )
  ->create();
```

Sử dụng các phương thức magic

Để thuận tiện, bạn có thể sử dụng các phương thức quan hệ factory ma thuật của Laravel để xây dựng mối quan hệ. Ví dụ: ví dụ sau sẽ sử dụng quy ước để xác định rằng các model liên quan nên được tạo thông qua phương thức quan hệ `posts` trên model `User`:

```
$user = User::factory()
  ->hasPosts(3)
  ->create();
```

Khi sử dụng các phương thức ma thuật để tạo mối quan hệ ban đầu, bạn có thể truyền một mảng thuộc tính attribute để ghi đè lên các model liên quan:

```
$user = User::factory()
  ->hasPosts(3, [
```

```
    'published' => false,
  ])
  ->create();
```

Bạn có thể cung cấp hàm xử lý chuyển đổi trạng thái nếu sự thay đổi trạng thái của bạn yêu cầu quyền truy cập vào model mẹ:

```
$user = User::factory()
  ->hasPosts(3, function (array $attributes, User $user) {
    return ['user_type' => $user->type];
  })
  ->create();
```

Mối quan hệ Belong To

Bây giờ chúng ta đã khám phá cách xây dựng mối quan hệ "has many" bằng cách sử dụng các factory, hãy cùng khám phá phần nghịch đảo của mối quan hệ. Phương thức `for` có thể được sử dụng để khai báo model mẹ mà các model lệ thuộc do factory tạo ra. Ví dụ: chúng ta có thể tạo ba đối tượng model `App\Models\Post` thuộc về một người dùng:

```
use App\Models\Post;
use App\Models\User;

$posts = Post::factory()
  ->count(3)
  ->for(User::factory()->state([
    'name' => 'Jessica Archer',
  ]))
  ->create();
```

Nếu bạn đã có một đối tượng model mẹ sẽ được liên kết với các model mà bạn đang tạo, bạn có thể truyền đối tượng model cho phương thức `for`:

```
$user = User::factory()->create();

$posts = Post::factory()
```

```
    ->count(3)
    ->for($user)
    ->create();
```

Sử dụng các phương thức magic

Để thuận tiện, bạn có thể sử dụng các phương thức quan hệ factory magic của Laravel để xác định các mối quan hệ "belongs to". Ví dụ: ví dụ sau sẽ sử dụng quy ước để xác định rằng ba bài đăng phải thuộc về mối quan hệ `user` trên model `Post`:

```
$posts = Post::factory()
    ->count(3)
    ->forUser([
        'name' => 'Jessica Archer',
    ])
    ->create();
```

Mối quan hệ many to many

Giống như mối quan hệ *has many*, mối quan hệ *"many to many"* có thể được tạo bằng cách sử dụng phương thức `has`:

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
    ->has(Role::factory()->count(3))
    ->create();
```

Các thuộc tính attribute của bảng pivot

Nếu bạn cần xác định các thuộc tính attribute cần được đặt trên bảng pivot/trung gian liên kết các model, bạn có thể sử dụng phương thức `hasAttached`. Phương thức này chấp nhận một mảng các tên và giá trị thuộc tính attribute của bảng trung gian làm đối số thứ hai của nó:

```php
use App\Models\Role;
use App\Models\User;

$user = User::factory()
  ->hasAttached(
    Role::factory()->count(3),
    ['active' => true]
  )
  ->create();
```

Bạn có thể cung cấp hàm xử lý chuyển đổi trạng thái nếu sự thay đổi trạng thái của bạn yêu cầu quyền truy cập vào model liên quan:

```php
$user = User::factory()
  ->hasAttached(
    Role::factory()
      ->count(3)
      ->state(function (array $attributes, User $user) {
        return ['name' => $user->name.' Role'];
      }),
    ['active' => true]
  )
  ->create();
```

Nếu bạn đã có các đối tượng model mà bạn muốn gắn vào các model mà bạn đang tạo, bạn có thể truyền các đối tượng model cho phương thức `hasAttached`. Trong ví dụ này, ba vai trò giống nhau sẽ được gắn cho cả ba người dùng:

```php
$roles = Role::factory()->count(3)->create();

$user = User::factory()
  ->count(3)
  ->hasAttached($roles, ['active' => true])
  ->create();
```

Sử dụng phương thức magic

Để thuận tiện, bạn có thể sử dụng các phương thức quan hệ nhà máy magic của Laravel để khai báo mối quan hệ *many to many*. Ví dụ: ví dụ sau sẽ sử dụng quy ước để xác định rằng các model liên quan nên được tạo thông qua phương thức quan hệ `roles` trên model `User`:

```
$user = User::factory()
  ->hasRoles(1, [
    'name' => 'Editor'
  ])
  ->create();
```

Các mối quan hệ đa hình

Các mối quan hệ đa hình cũng có thể được tạo ra bằng cách sử dụng factory. Mối quan hệ đa hình *"morph many"* được tạo ra theo cách tương tự như quan hệ *"has many"* điển hình. Ví dụ: nếu model `App\Models\Post` có mối quan hệ `morphMany` với model `App\Models\Comment`:

```
use App\Models\Post;

$post = Post::factory()->hasComments(3)->create();
```

Mối quan hệ *morph to*

Các phương pháp magic có thể không được sử dụng để tạo mối quan hệ `morphTo`. Thay vào đó, phương thức `for` phải được sử dụng trực tiếp và tên của mối quan hệ phải được cung cấp công khai. Ví dụ, hãy tưởng tượng rằng model `Comment` có một phương thức commentable xác định mối quan hệ `morphTo`. Trong trường hợp này, chúng tôi có thể tạo ba comment thuộc về một bài đăng bằng cách sử dụng trực tiếp phương thức `for`:

```
$comments = Comment::factory()->count(3)->for(
  Post::factory(), 'commentable'
)->create();
```

Mối quan hệ *many to many* đa hình

Mối quan hệ "many to many" (`morphToMany` / `morphedByMany`) đa hình có thể được tạo giống như mối quan hệ không đa hình "many to many":

```
use App\Models\Tag;
use App\Models\Video;

$videos = Video::factory()
  ->hasAttached(
    Tag::factory()->count(3),
    ['public' => true]
  )
  ->create();
```

Tất nhiên, phương thức magic `has` cũng có thể được sử dụng để tạo ra các mối quan hệ đa hình "many to many":

```
$videos = Video::factory()
  ->hasTags(3, ['public' => true])
  ->create();
```

Khai báo các mối quan hệ với các factory

Để tạo mối quan hệ trong factory model của bạn, bạn thường sẽ gán một đối tượng factory mới cho khóa ngoại của mối quan hệ. Điều này thường được thực hiện đối với các mối quan hệ "nghịch đảo" chẳng hạn như các mối quan hệ `belongsTo` và `morphTo`. Ví dụ: nếu bạn muốn tạo người dùng mới khi tạo bài đăng, bạn có thể làm như sau:

```
use App\Models\User;

/**
 * Define the model's default state.
 *
 * @return array
 */
```

```php
public function definition()
{
  return [
    'user_id' => User::factory(),
    'title' => $this->faker->title(),
    'content' => $this->faker->paragraph(),
  ];
}
```

Nếu các cột của mối quan hệ phụ thuộc vào factory khai báo nó, bạn có thể gán một hàm xử lý cho một thuộc tính attribute. Hàm xử lý sẽ nhận được mảng thuộc tính attribute đã đánh giá của factory:

```php
/**
 * Define the model's default state.
 *
 * @return array
 */
public function definition()
{
  return [
    'user_id' => User::factory(),
    'user_type' => function (array $attributes) {
      return User::find($attributes['user_id'])->type;
    },
    'title' => $this->faker->title(),
    'content' => $this->faker->paragraph(),
  ];
}
```

Chạy seeder

Nếu bạn muốn sử dụng chương trình gióng dữ liệu (*database seeder*) để đưa vào cơ sở dữ liệu của mình trong suốt quá trình kiểm tra tính năng, bạn có thể gọi phương thức `seed`. Theo mặc định, phương thức `seed` sẽ thực thi `DatabaseSeeder`, phương thức này sẽ thực thi tất cả các chương trình gióng dữ liệu khác của bạn. Ngoài ra, bạn truyền một tên class

gióng dữ liệu (seeder) cụ thể cho phương thức `seed`:

```php
<?php

namespace Tests\Feature;

use Database\Seeders\OrderStatusSeeder;
use Database\Seeders\TransactionStatusSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
  use RefreshDatabase;

  /**
   * Test creating a new order.
   *
   * @return void
   */
  public function test_orders_can_be_created()
  {
    // Run the DatabaseSeeder...
    $this->seed();

    // Run a specific seeder...
    $this->seed(OrderStatusSeeder::class);

    // ...

    // Run an array of specific seeders...
    $this->seed([
      OrderStatusSeeder::class,
      TransactionStatusSeeder::class,
      // ...
    ]);
```

```
    }
}
```

Ngoài ra, bạn có thể hướng dẫn Laravel tự động gióng dữ liệu vào cơ sở dữ liệu trước mỗi lần kiểm tra trong đó sử dụng trait `RefreshDatabase`. Bạn có thể thực hiện điều này bằng cách xác định thuộc tính `$seed` trên lớp thử nghiệm cơ sở của bạn:

```
<?php

namespace Tests;
use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;

    /**
     * Indicates whether the default seeder should run before each test.
     *
     * @var bool
     */
    protected $seed = true;
}
```

Khi thuộc tính `$seed` là `true`, kiểm tra sẽ chạy class `Database\Seeders\DatabaseSeeder` trước mỗi lần kiểm tra sử dụng trait `RefreshDatabase`. Tuy nhiên, bạn có thể chỉ định một seeder cụ thể sẽ được thực thi bằng cách khai báo thuộc tính `$seeder` trên class test của bạn:

```php
use Database\Seeders\OrderStatusSeeder;

/**
 * Run a specific seeder before each test.
 *
 * @var string
 */
protected $seeder = OrderStatusSeeder::class;
```

Assertion có sẵn

Laravel cung cấp một số xác nhận assertion các cơ sở dữ liệu cho các bài kiểm tra tính năng PHPUnit của bạn. Chúng ta sẽ thảo luận về từng xác nhận assertion này bên dưới.

assertDatabaseCount

Khẳng định rằng một bảng trong cơ sở dữ liệu chứa số lượng record đã cho:

```php
$this->assertDatabaseCount('users', 5);
```

assertDatabaseHas

Khẳng định rằng một bảng trong cơ sở dữ liệu chứa các record khớp với các ràng buộc truy vấn khóa/giá trị đã cho:

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

assertDatabaseMissing

Khẳng định rằng một bảng trong cơ sở dữ liệu không chứa các record khớp với các ràng buộc truy vấn khóa/giá trị đã cho:

```php
$this->assertDatabaseMissing('users', [
  'email' => 'sally@example.com',
```

```
]);
```

assertDeleted

`assertDeleted` khẳng định rằng một Eloquent model nhất định đã bị xóa khỏi cơ sở dữ liệu:

```
use App\Models\User;

$user = User::find(1);

$user->delete();

$this->assertDeleted($user);
```

Phương thức `assertSoftDeleted` có thể được sử dụng để xác nhận assertion một Eloquent model nhất định đã bị "xóa mềm":

```
$this->assertSoftDeleted($user);
```

assertModelExists

Khẳng định rằng một model nhất định tồn tại trong cơ sở dữ liệu:

```
use App\Models\User;

$user = User::factory()->create();

$this->assertModelExists($user);
```

assertModelMissing

Khẳng định rằng một model nhất định không tồn tại trong cơ sở dữ liệu:

```
use App\Models\User;
```

```php
$user = User::factory()->create();

$user->delete();

$this->assertModelMissing($user);
```