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

Database Testing

- # Introduction
- # Generating Factories
- # Resetting The Database After Each Test
- # Writing Factories
  - # Factory States
  - # Factory Callbacks
- # Using Factories
  - # Creating Models
  - # Persisting Models
  - # Relationships
- # Using Seeds
- # Available Assertions

Introduction

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

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

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

You can also use the `assertDatabaseMissing` helper to assert that data does not exist in the database.

The `assertDatabaseHas` 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.

Generating Factories

To create a factory, use the `make:factory` Artisan command:

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

The new factory will be placed in your `database/factories` directory.

The `--model` option may be used to indicate the name of the model created by the factory. This option will pre-fill the generated factory file with the given model:

```
php artisan make:factory PostFactory --model=Post
```

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. The `RefreshDatabase` trait takes the most optimal approach to migrating your test database depending on if you are using an in-memory database or a traditional database. Use the trait on your test class and everything will be handled for you:

```
<?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 testBasicExample()
    {
        $response = $this->get('/');

        // ...
    }
}
```

Writing Factories

When testing, you may 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 model factories. To get started, take a look at the `database/factories/UserFactory.php` file in your application. Out of the box, this file contains one factory definition:

```
use Faker\Generator as Faker;
use Illuminate\Support\Str;

$factory->define(App\User::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'email_verified_at' => now(),
        'password' => '$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdF
        '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.

You may also create additional factory files for each model for better organization. For example, you could create `UserFactory.php` and `CommentFactory.php` files within your `database/factories` directory. All of the files within the `factories` directory will automatically be loaded by Laravel.

You can set the Faker locale by adding a `faker_locale` option to your `config/app.php` configuration file.

Factory States

States allow you to define discrete modifications that can be applied to your model factories in any combination. For example, your `User` model might have a `delinquent` state that modifies one of its default attribute values. You may define your state transformations using the `state` method. For simple states, you may pass an array of attribute modifications:

```
$factory->state(App\User::class, 'delinquent', [
    'account_status' => 'delinquent',
]);
```

If your state requires calculation or a `$faker` instance, you may use a Closure to calculate the state's attribute modifications:

```
$factory->state(App\User::class, 'address', function ($faker) {
    return [
        'address' => $faker->address,
    ];
});
```

Factory Callbacks

Factory callbacks are registered using the `afterMaking` and `afterCreating` methods, and allow you to perform additional tasks after making or creating a model. For example, you may use callbacks to relate additional models to the created model:

```
$factory->afterMaking(App\User::class, function ($user, $faker) {
    // ...
```

```
});

$factory->afterCreating(App\User::class, function ($user, $faker) {
    $user->accounts()->save(factory(App\Account::class)->make());
});
```

You may also define callbacks for factory states:

```
$factory->afterMakingState(App\User::class, 'delinquent', function ($user, $faker)
    // ...
});

$factory->afterCreatingState(App\User::class, 'delinquent', function ($user, $fake
    // ...
});
```

Using Factories

Creating Models

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

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

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

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();
```

Applying States

You may also apply any of your states to the models. If you would like to apply multiple state transformations to the models, you should specify the name of each state you would like to apply:

```
$users = factory(App\User::class, 5)->states('delinquent')->make();

$users = factory(App\User::class, 5)->states('premium', 'delinquent')->make();
```

Overriding Attributes

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',
]);
```

Mass assignment protection is automatically disabled when creating models using factories.

Persisting 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()
{
    // Create a single App\User instance...
    $user = factory(App\User::class)->create();

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

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

```
}
```

You may override attributes on the model by passing an array to the `create` method:

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

Relationships

In this example, we'll attach a relation to some 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`:

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

You may use the `createMany` method to create multiple related models:

```
$user->posts()->createMany(
    factory(App\Post::class, 3)->make()->toArray()
);
```

Relations & Attribute Closures

You may also attach relationships to models in your factory definitions. For example, if you would like to create a new `User` instance when creating a `Post`, you may do the following:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => factory(App\User::class),
    ];
});
```

If the relationship depends on the factory that defines it you may provide a callback which accepts the evaluated attribute array:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => factory(App\User::class),
        'user_type' => function (array $post) {
            return App\User::find($post['user_id'])->type;
        },
    ];
});
```

Using Seeds

If you would like to use database seeders to populate your database during a test, you may use the `seed` method. By default, the `seed` method will return the `DatabaseSeeder`, which should execute all of your other seeders. Alternatively, you pass a specific seeder class name to the `seed` method:

```
<?php

namespace Tests\Feature;

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

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * Test creating a new order.
     *
     * @return void
```

```
     */
    public function testCreatingANewOrder()
    {
        // Run the DatabaseSeeder...
        $this->seed();

        // Run a single seeder...
        $this->seed(OrderStatusesTableSeeder::class);

        // ...
    }
}
```

Available Assertions

Laravel provides several database assertions for your PHPUnit tests:

Method	Description
`$this->assertDatabaseHas($table, array $data);`	Assert that a table in the database contains the given data.
`$this->assertDatabaseMissing($table, array $data);`	Assert that a table in the database does not contain the given data.
`$this->assertSoftDeleted($table, array $data);`	Assert that the given record has been soft deleted.

Our Partners

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.

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.