1v0 Ship v1.0 Faster

Testing Laravel Applications with PHPUnit

Testing Laravel Applications with PHPUnit

Testing is essential for building reliable, maintainable Laravel applications. With PHPUnit integrated out of the box, you can write unit and feature tests that validate your business logic, HTTP flows, and database interactions. This guide walks through setup, writing your first tests, and understanding typical test outputs.

Setting Up PHPUnit in Laravel

Laravel ships with PHPUnit preconfigured in `composer.json` under `require-dev`. Confirm the dependency and install vendor packages.

```
"require-dev": {
    "phpunit/phpunit": "^10.0"
}Code language: JSON / JSON with Comments (json)
```

This ensures PHPUnit is available to Laravel's test runner. If you're upgrading, run `composer update` to get the right version for your PHP/Laravel stack.

```
composer install
php artisan testCode language: Bash (bash)
```

The first run should execute the default example test and report results in a readable, colored format.

Creating Your First Unit Test

Use Artisan to scaffold a unit test class. Unit tests focus on small, isolated pieces of logic without the framework bootstrapping overhead.

```bash
php artisan make:test UserMathTest --unit
```
Code language: Bash (bash)

Open the generated file at `tests/Unit/UserMathTest.php` and add a simple assertion.

```php
namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class UserMathTest extends TestCase
{
    public function test_basic_math_addition()
    {
        $this->assertSame(4, 2 + 2);
    }
}
```
Code language: PHP (php)

This basic test verifies your PHPUnit setup. It runs quickly and proves your environment is configured correctly.

Writing a Feature Test (HTTP + Database)

Feature tests exercise full request lifecycles, including routes, controllers, middleware, and the database. Let’s test a simple “create post” flow.

```bash
php artisan make:test PostCreationTest
```
Code language: Bash (bash)

Update `tests/Feature/PostCreationTest.php` to validate the HTTP response and database changes.

```php
namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;
use App\Models\User;
use App\Models\Post;

class PostCreationTest extends TestCase
{
    use RefreshDatabase;

    public function test_authenticated_user_can_create_post()
    {
        $user = User::factory()->create();

        $response = $this->actingAs($user)
            ->post('/posts', [
                'title' => 'My First Post',
                'body'  => 'Hello world!',
            ]);

        $response->assertRedirect('/posts');
        $this->assertDatabaseHas('posts', [
            'title' => 'My First Post',
        ]);
    }
}
```
Code language: PHP (php)

This test signs in a user, posts form data, expects a redirect, and verifies that the record exists. For production-grade examples of auth flows, also see How to Build Email Verification in Laravel 12 (Step by Step) and Implementing Password Reset in Laravel 12 Without Packages.

Running Tests and Filtering by Class/Method

Run the entire test suite, a single class, or even a single method using filters.

```bash
# run all tests
php artisan test

# run a specific class
php artisan test --filter=PostCreationTest

# run a specific test method
php artisan test --filter=PostCreationTest::test_authenticated_user_can_create_post
```
Code language: Bash (bash)

Filtering speeds up the feedback loop while you iterate on a failing scenario.

Sample Output from `php artisan test`

Here’s a typical output when all tests pass. Your numbers will differ based on how many tests/assertions you have.

```bash
PHPUnit 10.*/Laravel Test Runner

   PASS  Tests\Unit\UserMathTest
  ✓ test_basic_math_addition

   PASS  Tests\Feature\PostCreationTest
  ✓ test_authenticated_user_can_create_post

  Tests:  2 passed
  Assertions: 3
  Time: 0.58s
```
Code language: Bash (bash)

When a test fails, the output highlights the failing assertion, expected vs actual values, and a snippet of the stack trace to help you navigate the source quickly.

Seeding, Factories, and Faster Test Data

Use model factories and seeders to generate realistic test data quickly. This keeps tests readable and reduces duplication across scenarios.

```
// Example: creating many posts for a listing test
$posts = \App\Models\Post::factory()->count(10)->create();Code language: PHP (php)
```

Factories keep your tests expressive and focused on behavior. For deeper coverage, see Using Laravel Factories and Seeders for Test Data.

UI-Related Testing Tips

For Blade-driven UIs, test the rendered output and important view data. Focus on what matters (HTTP status, redirected routes, session flashes, and presence of key strings).

```
public function test_posts_index_renders_titles()
{
    $posts = \App\Models\Post::factory()->count(2)->create([
        'title' => 'Visible In List'
    ]);

    $response = $this->get('/posts');
    $response->assertOk()
        ->assertSee('Visible In List');
}
```
Code language: PHP (php)

This checks that the index page renders properly and contains expected text. For browser-level interactions (clicks, JS), consider How to Use Laravel Dusk for Browser Testing.

Wrapping Up

You learned how to confirm your PHPUnit setup, create unit and feature tests, filter test runs, read outputs, and leverage factories for fast data setup. A disciplined testing habit yields fewer regressions and more confidence when refactoring.

What's Next

Continue strengthening your test suite with these related guides:

- How to Write Feature Tests in Laravel for APIs
- Using Laravel Factories and Seeders for Test Data
- How to Use Laravel Dusk for Browser Testing