ScopeTreasure Courses

LARAVEL COURSES · LESSON LV.19 · CODE SCOPE SERIES

Testing Ki Bunyad — Neend Sukoon Ki

Guards for your code itself: feature vs unit tests, factories for fake students, actingAs, RefreshDatabase — and a red test that catches a real bug.

Introduction

Every lesson so far ended the same way: *you* clicked around and checked the app by hand. That works — until the app has forty pages and your "quick check" misses the one page your last change broke, which a real user finds first. **Tests** are those hand-checks written down as code: a robot visitor who clicks the important paths after every change and reports green or red in seconds.

Two kinds live here: **unit tests** examine a small piece alone (does this calculation behave?), and **feature tests** walk whole journeys (can a guest reach the students page? does the form save?). This lesson writes the Students Manager's first feature tests — with factories inventing fake students and `RefreshDatabase` handing every test a spotless database — and it does the bravest thing a testing lesson can: breaks something on purpose and lets a red test catch it.

(Roman Urdu: Haath se jaanchna aaj kaam aata hai; test likhna kal ki neend hai — robot har tabdeeli ke baad khud chakkar laga kar report dega.)

Key points

- **A test is a promise with a guard:** each test says "this must ALWAYS be true" — guests bounce off /students, valid forms save, invalid marks error. When a future change breaks a promise, the test turns red BEFORE a user finds out.
- **Feature vs unit:** feature tests drive the app through its real doors (URLs, forms, database) and cover most of a Laravel app's risks — start there. Unit tests isolate one class or formula; fewer, sharper, faster. `php artisan make:test StudentsTest` creates a feature test; `--unit` makes the other kind.
- **Run everything with `php artisan test`:** the whole suite executes in seconds and reports per-test ticks and a summary — `Tests: 6 passed`. Green is a fact, not a feeling.
- **Assertions are the verbs of checking:** `$response->assertOk()` (200), `->assertRedirect('/login')`, `->assertSee('Ali Raza')`, `$this->assertDatabaseHas('students', [...])`, `->assertSessionHasErrors('marks')`. Each names exactly one expectation — a red test then tells you WHICH promise broke.

  (Roman Urdu: Assertion matlab saaf daawa — "ye hona chahiye tha". Laal nateeja phir ye bhi batata hai ke daawa kahan toota.)
- **Factories invent citizens:** `Student::factory()->count(10)->create()` fills the test database with believable fake students from a recipe (`StudentFactory`) — no hand-typed rows, no touching real data.
- **`actingAs($user)` walks in as someone:** LV.7's login, performed by the robot: `$this->actingAs($teacher)->get('/students')`. With it you test both sides of every guard — guest, teacher,

admin (LV.18's three fates, automated).

- `RefreshDatabase` **is the clean slate:** the trait rebuilds the tables fresh for each test (on a separate TEST database — your `.env.testing` or in-memory SQLite), so tests never pollute each other or your real data.

Code examples

Example 1 — The guard promises: guest bounces, teacher enters

```
php artisan make:test StudentsAccessTest
```

```
// tests/Feature/StudentsAccessTest.php
use Illuminate\Foundation\Testing\RefreshDatabase;

class StudentsAccessTest extends TestCase
{
    use RefreshDatabase;

    public function test_guests_cannot_see_students(): void
    {
        $response = $this->get('/students');

        $response->assertRedirect('/login');
    }

    public function test_teacher_sees_the_students_list(): void
    {
        Student::factory()->create(['name' => 'Ali Raza', 'marks' => 88]);
        $teacher = User::factory()->create();

        $response = $this->actingAs($teacher)->get('/students');

        $response->assertOk();
        $response->assertSee('Ali Raza');
    }
}
```

What it does: Two of LV.7's hand-checks, written forever. `RefreshDatabase` rebuilds the empty tables; the factory plants one student; the robot visits as guest (expects the login redirect) and as teacher (expects 200 and Ali's name on the page).

What you will see: `php artisan test` runs the file and reports its promises kept:

```
   PASS  Tests\Feature\StudentsAccessTest
  ✓ guests cannot see students
  ✓ teacher sees the students list
```

Example 2 — The form promises: valid saves, invalid bounces

```
public function test_teacher_can_add_a_student(): void
{
    $teacher = User::factory()->create();

    $response = $this->actingAs($teacher)->post('/students', [
        'name'  => 'Sara Khan',
        'city'  => 'Lahore',
        'marks' => 92,
    ]);

    $response->assertRedirect(route('students.index'));
    $this->assertDatabaseHas('students', ['name' => 'Sara Khan', 'marks' => 92]);
}

public function test_marks_above_100_are_rejected(): void
{
    $teacher = User::factory()->create();

    $response = $this->actingAs($teacher)->post('/students', [
        'name'  => 'Test Student',
        'city'  => 'Sahiwal',
        'marks' => 150,
    ]);

    $response->assertSessionHasErrors('marks');
    $this->assertDatabaseMissing('students', ['name' => 'Test Student']);
}
```

What it does: LV.5's validation, guarded permanently. The robot POSTs like a browser (CSRF handled by the test framework), then checks BOTH the redirect and the actual table — a page can lie; the database cannot. The invalid case asserts the mirror image: errors exist AND nothing was saved.

What you will see: two more green ticks. Remove the `max:100` rule from the controller for one mischievous minute and run again — the second test turns red, naming the broken promise. Restore the rule; green returns. You have just watched a test earn its salary.

Example 3 — The red test that catches a real bug

```
// the class report helper under test (app/Support/ClassReport.php)
public static function average(): float
{
    return Student::avg('marks') ?? 0;
}
```

```
public function test_class_average_matches_hand_calculation(): void
{
    Student::factory()->create(['marks' => 88]);
    Student::factory()->create(['marks' => 92]);
    Student::factory()->create(['marks' => 76]);
    Student::factory()->create(['marks' => 95]);
    Student::factory()->create(['marks' => 64]);

    $this->assertEquals(83.0, \App\Support\ClassReport::average());
}
```

What it does: The series' own five students, hand-averaged in LV.10 to exactly 83, become the permanent expected answer. Now imagine a tired evening's edit makes `average()` secretly average only marks above 70 — the robot disagrees within seconds:

What you will see: the failure, honest and specific:

```
   FAIL  Tests\Feature\ClassReportTest
  × class average matches hand calculation
  Failed asserting that 87.75 matches expected 83.0.
```

87.75 is the telltale (351 ÷ 4 — Usman's 64 quietly excluded). Fix the helper; the suite returns to green. That is the whole culture of testing in one cycle: **write the promise, watch it catch a lie, fix the code — never the promise.**

Quick recap

- Tests are hand-checks written as code: run `php artisan test` after every change; green is a fact.
- Feature tests walk real journeys; unit tests isolate pieces. Assertions name one expectation each.
- Factories invent data, `actingAs` walks in as any user, `RefreshDatabase` keeps tests spotless and separate.
- A red test is good news delivered early — fix the code, not the promise.

Common mistakes

1. **Testing only the happy path** — a suite where everyone is logged in and every form is valid proves the sunny day only. Fix: for every door, test the guest, the wrong data, and the forbidden role alongside the success.
2. **Tests touching the real database** — a test run that deletes or pollutes your development data (or worse). Fix: `RefreshDatabase` plus a dedicated test database (in-memory SQLite is perfect) — isolation is not optional.
3. **"Fixing" a red test by weakening it** — changing the expected 83 to 87.75 because the code says so. Fix: the test encodes the PROMISE (hand-checked truth); when they fight, the code is wrong until proven otherwise.
4. **One giant test doing everything** — a single red then points nowhere. Fix: one promise per test, named like a sentence (`test_marks_above_100_are_rejected`) — the failing name IS the diagnosis.

Practice

1. Write the delete promises: a teacher can delete a student (row gone, `assertDatabaseMissing`), and a guest's delete attempt lands on /login. Then protect destroy with LV.18's `role:admin` and update the suite so the teacher case now expects 403 — watch the suite guide your hand.
2. Add a factory state: `Student::factory()->topper()` producing marks between 90 and 100 (a `state()` in the factory). Use it to write `test_merit_list_shows_only_toppers` against LV.10's scope page.

Practice quiz

1. What is the difference between feature tests and unit tests?

- There is none
- Feature tests drive the app through its real doors (URLs, forms, database) and cover most risks, start there; unit tests isolate one class or formula, fewer, sharper, faster
- Feature tests are slower and useless
- Unit tests drive the whole app

Answer: Feature tests drive the app through its real doors (URLs, forms, database) and cover most risks, start there; unit tests isolate one class or formula, fewer, sharper, faster

2. How do you run the whole Laravel test suite, and what does green mean?

- php artisan serve, green means the server runs
- php artisan test, the suite executes in seconds and reports per-test ticks and a summary like Tests: 6 passed; green is a fact, not a feeling
- By hand-clicking every page
- Tests cannot be run, only written

Answer: php artisan test, the suite executes in seconds and reports per-test ticks and a summary like Tests: 6 passed; green is a fact, not a feeling

3. How do tests get believable data and walk in as any user, without polluting real data?

- By hand-typing rows into the real database
- Factories invent citizens with Student::factory()->count(10)->create(), actingAs($user) walks in as someone, and RefreshDatabase rebuilds the tables fresh for each test on a separate test database
- Tests always use real data
- Data is not needed for tests

Answer: Factories invent citizens with Student::factory()->count(10)->create(), actingAs($user) walks in as someone, and RefreshDatabase rebuilds the tables fresh for each test on a separate test database

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/laravel-testing · More free tools and notes at https://scopetreasure.com