1v0 Ship v1.0 Faster

How to Write Feature Tests in Laravel for APIs

How to Write Feature Tests in Laravel for APIs

Feature tests validate full request lifecycles—routes, middleware, controllers, policies, database, and JSON responses. In this guide, you'll create API endpoints, secure them, and write expressive feature tests that assert status codes, payload structure, and side effects.

Prerequisites and Setup

Ensure you have a working API stack and database test environment. If you're building a token-based API, consider first reading How to Build a REST API with Laravel 12 & Sanctum and Securing Laravel APIs with Sanctum: Complete Guide.

```
php artisan make:model Post -mf
php artisan make:controller Api/PostController --api
php artisan make:test Api/PostApiTestCode language: Bash (bash)
```

This scaffolds a `Post` model, migration, factory, an API controller, and a feature test class to exercise the HTTP layer.

Create Routes and Controller Methods

Define minimal endpoints for listing and creating posts. These examples assume you'll protect `store` with auth middleware (e.g., Sanctum).

```
// routes/api.php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\Api\PostController;

Route::get('/posts', [PostController::class, 'index']);
Route::middleware('auth:sanctum')->post('/posts',
[PostController::class, 'store']);Code language: PHP (php)
```

`GET /posts` is public for demonstration; `POST /posts` requires authentication via `auth:sanctum`. Adjust to your security needs.

```
// app/Http/Controllers/Api/PostController.php
namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;
use Illuminate\Http\Response;

class PostController extends Controller
{
    public function index()
    {
        return response()->json([
            'data' =>
Post::latest()->select(['id','title','body'])->paginate(10)
        ]);
    }

    public function store(Request $request)
    {
        $validated = $request->validate([
            'title' => ['required','string','max:120'],
            'body'  => ['required','string'],
        ]);

        $post = Post::create($validated);
```

```
        return response()->json([
            'message' => 'Created',
            'data'    => [
                'id'    => $post->id,
                'title' => $post->title,
                'body'  => $post->body,
            ],
        ], Response::HTTP_CREATED);
    }
}Code language: PHP (php)
```

The controller returns predictable JSON structures so your tests can assert both shape and content. For clean response formatting in larger projects, also see How to Use Eloquent API Resources for Clean APIs.

```
// database/migrations/xxxx_xx_xx_create_posts_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up(): void {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }
    public function down(): void {
        Schema::dropIfExists('posts');
    }
};Code language: PHP (php)
```

This migration creates a minimal `posts` table. Keep columns concise to make focused tests faster and more reliable.

```
// database/factories/PostFactory.php
namespace Database\Factories;
```

```
use Illuminate\Database\Eloquent\Factories\Factory;

class PostFactory extends Factory
{
    public function definition(): array
    {
        return [
            'title' => $this->faker->sentence(6),
            'body'  => $this->faker->paragraph(),
        ];
    }
}Code language: PHP (php)
```

Factories provide quick, expressive test data. For deeper patterns, see Using Laravel Factories and Seeders for Test Data.

Writing Feature Tests for API Endpoints

Below is the complete test suite for a small Posts API. Right after the code, you'll find a line-by-line breakdown explaining every import, trait, method, assertion, and why it matters.

```
// tests/Feature/Api/PostApiTest.php
namespace Tests\Feature\Api;

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

class PostApiTest extends TestCase
{
    use RefreshDatabase;
```

```
    public function test_can_list_posts_as_json()
    {
        Post::factory()->count(3)->create();

        $response = $this->getJson('/api/posts');

        $response->assertOk()
                 ->assertJsonStructure(['data' => ['data' =>
[['id','title','body']]]]);
    }

    public function test_cannot_create_post_when_unauthenticated()
    {
        $response = $this->postJson('/api/posts', [
            'title' => 'Hello',
            'body'  => 'World',
        ]);

        $response->assertUnauthorized();
    }

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

        $response = $this->actingAs($user, 'sanctum')
                         ->postJson('/api/posts', [
                             'title' => 'My Title',
                             'body'  => 'Body text',
                         ]);

        $response->assertCreated()
                 ->assertJsonPath('data.title', 'My Title');

        $this->assertDatabaseHas('posts', ['title' => 'My Title']);
    }

    public function test_validation_errors_are_returned()
```

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

        $response = $this->actingAs($user, 'sanctum')
                         ->postJson('/api/posts', [
                             'title' => '',   // invalid
                             'body'  => '',   // invalid
                         ]);

        $response->assertStatus(422)
                 ->assertJsonValidationErrors(['title','body']);
    }
}Code language: PHP (php)
```

Namespace & Imports

- `namespace Tests\Feature\Api;` — Organizes this test under `Feature/Api` so it's discoverable by PHPUnit and matches your folder structure.
- `use Tests\TestCase;` — Base Laravel test case boots the application, loads the HTTP kernel, and gives you helpers like `getJson`, `postJson`, and `actingAs`.
- `use App\Models\User;` and `use App\Models\Post;` — Import Eloquent models you use in factories and database assertions.
- `use Illuminate\Foundation\Testing\RefreshDatabase;` — Trait that wraps each test in a transaction or re-runs migrations to ensure a clean database for isolation and reliability.

Class & Trait

- `class PostApiTest extends TestCase` — Extends the Laravel-aware `TestCase` so your app container, routing, middleware, and database are available.
- `use RefreshDatabase;` — Ensures each test starts with a known DB state (empty tables unless your migrations seed).

Test 1: Listing Posts as JSON

- `Post::factory()->count(3)->create();` — Seeds three posts using your factory so the index endpoint has data to return.
- `$this->getJson('/api/posts');` — Performs a JSON GET request to your API route, automatically setting the `Accept: application/json` header.

- `$response->assertOk();` — Expects HTTP 200; ensures the route is registered and didn't error.
- `assertJsonStructure(['data' => ['data' => [[ 'id','title','body' ]]]])` — Validates the JSON shape. The duplication of `data` is intentional if you're wrapping a Laravel paginator: the outer `data` is your envelope; the inner `data` is the paginated array of items. Each item must include `id`, `title`, and `body`.

Test 2: Unauthenticated Create Should Fail

- `$this->postJson('/api/posts', [...])` — Sends a JSON POST to the protected endpoint without credentials.
- `->assertUnauthorized();` — Expects HTTP 401, which implies your route is correctly behind `auth:sanctum` (or equivalent).
- Purpose: Ensures that unauthenticated users can't create resources, protecting data integrity.

Test 3: Authenticated Create Should Succeed

- `$user = User::factory()->create();` — Creates a real user record to authenticate requests against.
- `$this->actingAs($user, 'sanctum')` — Authenticates as `$user` using the `sanctum` guard so the request passes the `auth:sanctum` middleware.
- `->postJson('/api/posts', ['title' => 'My Title','body' => 'Body text'])` — Submits valid payload to create a post.
- `$response->assertCreated();` — Expects HTTP 201 Created, matching REST semantics in your controller's `store` method.
- `->assertJsonPath('data.title','My Title');` — Reads a specific JSON path and asserts the value, confirming the API echoes created data consistently.
- `$this->assertDatabaseHas('posts',['title' => 'My Title']);` — Verifies the side effect (DB write) happened correctly, not just the HTTP layer.

Test 4: Validation Errors Are Returned

- `$this->actingAs($user, 'sanctum')` — Ensures we're testing pure validation, not auth.
- `->postJson('/api/posts', ['title' => '', 'body' => ''])` — Sends invalid input to trigger the validator.
- `$response->assertStatus(422);` — 422 Unprocessable Entity is the expected status for validation failures in JSON APIs.

- `->assertJsonValidationErrors(['title','body']);` — Confirms the error bag contains keys for the invalid fields, which your frontend can render inline.

Why `getJson`/`postJson` Instead of `get`/`post`? These helpers automatically set JSON headers, ensuring your app returns JSON responses (e.g., validation error format) instead of redirecting with HTML. This keeps tests deterministic and representative of real API clients.

Why `RefreshDatabase`? It guarantees each test runs against a clean schema and data set. This prevents hidden coupling between tests (e.g., leftovers from a previous test) and makes failures reproducible.

Why Assert Both HTTP and Database? HTTP assertions confirm routing, middleware, and controller logic, while database assertions confirm side effects. Together, they validate behavior end to end.

With these explanations, you can confidently modify routes, policies, validation rules, and response formats while keeping your tests expressive and trustworthy.

Sample Test Output

Running the suite with `php artisan test` should show all green when your endpoints and tests align.

```
PHPUnit 10.*/Laravel Test Runner

   PASS  Tests\Feature\Api\PostApiTest
  ✓ test_can_list_posts_as_json
  ✓ test_cannot_create_post_when_unauthenticated
  ✓ test_authenticated_user_can_create_post
  ✓ test_validation_errors_are_returned

  Tests:  4 passed
  Assertions: 11
```

```
  Time: 0.89sCode language: Bash (bash)
```

For a failing sample (e.g., missing auth), PHPUnit highlights the failing expectation with a diff of expected vs. actual, plus a stack trace line to jump into your test file and controller.

Advanced Tips: Policies, Rate Limits, and Resources

Layer in authorization and rate limiting for production realism. Use policies to gate actions and add tests that assert `403` for forbidden operations. For token issuance and scopes, consider How to Add JWT Authentication to Laravel APIs if JWT fits your architecture.

```
// Example: assert rate limiting headers exist (if enabled)
$response = $this->getJson('/api/posts');
$response->assertOk();
$this->assertTrue($response->headers->has('X-RateLimit-
Remaining'));Code language: PHP (php)
```

These checks help verify production-grade API behavior beyond basic CRUD: authorization, throttling, and predictable resource shapes.

Wrapping Up

You built API endpoints and wrote comprehensive feature tests covering listing, authentication, creation, and validation. You also learned how to read test output and assert headers and JSON shapes—key skills for keeping APIs robust during rapid iteration.

What’s Next

Deepen your API testing and security with these related guides:

- How to Build a REST API with Laravel 12 & Sanctum
- Securing Laravel APIs with Sanctum: Complete Guide
- How to Use Eloquent API Resources for Clean APIs