Testing with Pest — cookbook

Setup, factories, feature, browser, fakes, time, arch, CI

1 · Setup and factories

```
<!-- phpunit.xml — tests get their OWN services -->
<env name="APP_ENV" value="testing"/>
<env name="BCRYPT_ROUNDS" value="4"/>
<env name="CACHE_STORE" value="array"/>
<env name="DB_CONNECTION" value="sqlite"/>
<env name="DB_DATABASE" value=":memory:"/>
<env name="MAIL_MAILER" value="array"/>
<env name="QUEUE_CONNECTION" value="sync"/>
<env name="SESSION_DRIVER" value="array"/>
```

```
// tests/Pest.php
uses(Tests\TestCase::class, LazilyRefreshDatabase::class)->in('Feature');
```

```
// a factory models the STORY, not just a row — states + relations as intent
final class OrderFactory extends Factory {
    public function definition(): array {
        return ['customer_id' => Customer::factory(), 'status' => 'draft',
                'total_cents' => 0, 'placed_at' => null, 'paid_at' => null];
    }
    public function placed(): static {
        return $this->state(fn () => ['status' => 'placed', 'placed_at' => now()]);
    }
    public function paid(): static {
        return $this->placed()->state(fn () => ['status' => 'paid', 'paid_at' => now()]);
    }
    public function withLine(Product $p, int $qty = 1): static {
        return $this->afterCreating(fn (Order $o) => $o->lines()->create([
            'product_id' => $p->id, 'quantity' => $qty, 'unit_price_cents' => $p->price_cents]));
    }
}

// the call reveals the history
$order = Order::factory()->paid()->withLine($product, quantity: 2)->create();
```

2 · A feature test = the HTTP contract

```
// validation at the HTTP boundary; the action performs the use case
final class StoreAccountOrderRequest extends FormRequest {
    public function authorize(): bool {
        return $this->user()?->can('createOrder', $this->route('account')) ?? false;
    }
    public function rules(): array {
```

```
        return ['product_sku' => ['required', 'string', 'exists:products,sku'],
                'quantity' => ['required', 'integer', 'min:1', 'max:500']];
    }
}
```

```
it('rejects an unorderable product with 422', function (): void {
    $account = Account::factory()->create();
    $user = User::factory()->create();
    $account->users()->attach($user, ['role' => 'buyer']);
    $product = Product::factory()->create(['is_orderable' => false]);

    $this->actingAs($user)
        ->postJson(route('accounts.orders.store', $account), [
            'product_sku' => $product->sku, 'quantity' => 3,
        ])
        ->assertStatus(422)
        ->assertJsonValidationErrors('product_sku');
});
```

Authorization and validation are **separate behaviours** — test each. The controller is a translation layer Request → Action → Resource.

3 · A browser test = a journey, not a screen inventory

```
beforeEach(function (): void {
    Http::preventStrayRequests();                    // no test goes to the network
    $this->customer = User::factory()->create();
    $this->billingCategory = SupportCategory::factory()->create(['name' => 'Billing', 'is_active' =>
true]);
});

it('lets a customer submit a support request', function (): void {
    $this->actingAs($this->customer);

    visit(route('support-requests.create'))
        ->assertSee('New support request')
        ->assertNoJavaScriptErrors()
        ->fill('Subject', 'Invoice contains the wrong company name')
        ->select('Category', $this->billingCategory->name)
        ->fill('Message', 'Please correct the name before the deadline.')
        ->click('Send request')
        ->waitForText('Your request has been sent')       // wait for the CUSTOMER-visible outcome
        ->assertNoJavaScriptErrors();

    expect(SupportRequest::query()->where([
        'user_id' => $this->customer->id, 'subject' => 'Invoice contains the wrong company name',
    ])->exists())->toBeTrue();
});
```

Select controls as a user does (label, text), create all state explicitly, cover **one** recoverable failure (e.g. a missing required field → the form stays).

4 · Fakes, time, external boundaries

```
Event::fake([OrderPaid::class]);      // fake NARROWLY — only this event
Event::assertDispatched(OrderPaid::class, fn (OrderPaid $e) => $e->order->is($order));

Mail::fake();  Queue::fake();
Mail::assertQueued(OrderReceipt::class, fn ($m) => $m->hasTo($order->customer->email));
Queue::assertPushed(GenerateInvoicePdf::class, fn ($j) => $j->order->is($order));

Storage::fake('invoices');
Storage::disk('invoices')->assertExists("{$order->id}/invoice.pdf");

Http::preventStrayRequests();
Http::fake(['https://accounting.example.test/api/invoices' => Http::response(['id' => 'inv_123'],
201)]);
Http::assertSent(fn (Request $r) => $r->url() === '…/api/invoices' && $r['reference'] === $order-
>external_reference);

// time — around a business boundary
$this->travelTo(Carbon::parse('2026-09-06 10:00:00'));
$link = DownloadLink::factory()->create(['expires_at' => now()->addMinutes(15)]);
$this->travel(16)->minutes();
$this->get(route('downloads.show', $link))->assertGone();
$this->travelBack();
```

> **COUNTEREXAMPLE** Keep one *integration* test that runs the real path (payment → listener → job dispatch) without fakes where the connection is what matters.

5 · Architecture tests + CI

```
arch('HTTP controllers are thin')
    ->expect('App\\Http\\Controllers')
    ->toOnlyUse(['App\\Http\\Requests', 'App\\Http\\Resources', 'App\\Actions', 'Illuminate']);

arch('domain code does not depend on infrastructure')
    ->expect('App\\Domain')
    ->not->toUse('App\\Infrastructure')
    ->not->toUse('Illuminate\\Support\\Facades');
```

```
# CI: cheap deterministic checks first, then behaviour
quality:   vendor/bin/pint --test  +  phpstan analyse   (matrix PHP 8.4/8.5)
tests:     php artisan test --compact --parallel --shard=${{ matrix.shard }}/4
browser-contract:  php artisan test --compact --group=browser-contract
```

Each shard = a disjoint slice; branch protection requires **every** one. Coverage is a *question* — why wasn't this branch executed — not a numeric target. Generate the HTML report while investigating a change:

```
XDEBUG_MODE=coverage php artisan test --compact --coverage-html=build/coverage
```

BEFORE YOU ENFORCE Let the team see a few failures and agree on the fix path. A controller rule needs an obvious action to move work into; a flaky parallel test needs a reproducible fixture fix. Keep the rule, the command and the error message close together.