THE RESILIENT LARAVEL CHECKLIST

12 patterns with practical Laravel code examples

Build software that keeps working
when things go wrong.

You cannot control when an external service fails.
You can control how your application responds.

FOR PHP & LARAVEL DEVELOPERS

Review your integrations. Protect critical flows.
Build systems your users can trust.

Marcelo Nizato
Resilient Developers
marcelonizato.dev

Start with one integration.

Choose a critical dependency. Check a box only when the behavior is implemented and verified. The pattern pages show a starting point and a failure test.

Integration: _____ Owner: _____

- [ ] 01. Set explicit timeouts / 03
  Every remote call has a connection and total request timeout.
- [ ] 02. Retry selectively, with jitter / 04
  Retries are bounded and limited to safe, transient failures.
- [ ] 03. Make payment writes idempotent / 05
  One business operation keeps one persisted idempotency key.
- [ ] 04. Deduplicate webhook effects / 06
  Authenticated event IDs are unique and applied atomically.
- [ ] 05. Move deferrable work to queues / 07
  Background jobs have bounded attempts and recovery ownership.
- [ ] 06. Open a circuit during outages / 08
  A failing dependency can be skipped, then probed safely.
- [ ] 07. Isolate dependency capacity / 09
  One busy integration cannot consume every worker.
- [ ] 08. Serve an honest fallback / 10
  Read-only features can degrade without inventing success.
- [ ] 09. Validate external contracts / 11
  Unexpected payloads stop at the integration boundary.
- [ ] 10. Persist intent with an outbox / 12
  A committed business change cannot lose its delivery intent.
- [ ] 11. Make failures observable / 13
  Each integration call records safe, correlated evidence.
- [ ] 12. Test the unhappy paths / 14
  Failure behavior is verified before a provider outage.

First action: _____ Review date: _____

01

Set explicit timeouts

A slow provider can occupy the workers your customers need. Give each dependency a time budget that fits the entire user request.

```
use Illuminate\Support\Facades\Http;
use Illuminate\Http\Client\ConnectionException;

try {
    $response = Http::connectTimeout(2)->timeout(5)
        ->get(config('services.catalog.url').'/items');
    $response->throw();
} catch (ConnectionException $e) {
    return response()->json([
        'message' => 'Catalog temporarily unavailable.',
    ], 503);
}
```

IMPLEMENTATION NOTES

This read-only example handles transport failures. Handle HTTP failures separately. For payment writes, a timeout means the outcome is unknown: reconcile with the provider before trying a new operation.

PROVE IT UNDER FAILURE

Simulate a connection failure. Confirm the caller gets a controlled response and no unbounded wait. Use a slow local stub to verify real elapsed time.

CHECK / Every remote call has a connection and total request timeout.

Framework references: 1 (page 16).

02

Retry selectively, with jitter

A retry may recover a brief outage. A crowd of immediate retries can amplify it. Choose one retry layer and budget its total time.

```
use Illuminate\Support\Facades\Http;
use Illuminate\Http\Client\ConnectionException;
use Illuminate\Http\Client\RequestException;

$response = Http::connectTimeout(1)->timeout(2)
    ->retry(3,
        fn (int $attempt) =>
            100 * (2 ** ($attempt - 1)) + random_int(0, 100),
        fn ($e) => $e instanceof ConnectionException
            || ($e instanceof RequestException
                && in_array($e->response->status(),
                    [502, 503, 504], true))
    )->get(config('services.catalog.url').'/items');
```

IMPLEMENTATION NOTES

Three attempts means at most two retries. This example retries a GET, excludes 429, and may take roughly 6.5 seconds plus overhead. Handle exhaustion. For 429, respect Retry-After with a bounded deferred retry.

PROVE IT UNDER FAILURE

Return 503, 503, then 200: expect three calls. Return 422: expect one. Do not multiply HTTP retries by unbounded job retries.

CHECK / Retries are bounded and limited to safe, transient failures.

Framework references: 1 (page 16).

03

Make payment writes idempotent

A customer clicks twice. A worker restarts after the provider charges the card. Replaying the same operation must not create a second charge.

```
use Illuminate\Support\Facades\Http;

// $payment is a persisted, authorized payment attempt.
// idempotency_key is immutable and unique in your DB.
$response = Http::connectTimeout(2)->timeout(5)
    ->withHeaders([
        'Idempotency-Key' => $payment->idempotency_key,
    ])->post(config('services.payments.url').'/charges', [
        'amount' => $payment->amount_minor,
        'currency' => $payment->currency,
        'reference' => (string) $payment->id,
    ])->throw();
```

IMPLEMENTATION NOTES

The provider must support this contract. Bind the key to the same immutable payload and account; keep it across all retries. On timeout, retain pending/unknown status and query by reference. A local lock alone cannot prevent remote duplicates.

PROVE IT UNDER FAILURE

Replay after a simulated lost response. Verify a single provider charge. Test key expiry and payload mismatch against the provider sandbox.

CHECK / One business operation keeps one persisted idempotency key.

Framework references: 1, 5 (page 16).

04

Deduplicate webhook effects

Providers redeliver events. Store receipt and local effects in one database transaction so a crash does not leave an event marked done without its effect.

```
use Illuminate\Support\Facades\DB;

// After signature, timestamp and payload verification.
// UNIQUE(provider, event_id) on webhook_receipts.
DB::transaction(function () use ($event) {
    DB::table('webhook_receipts')->insert([
        'provider' => 'payments',
        'event_id' => $event['id'],
    ]);
    DB::table('payments')
        ->where('provider_id', $event['payment_id'])
        ->where('status', 'pending')
        ->update(['status' => 'paid']);
});
```

IMPLEMENTATION NOTES

Example: a verified payment-succeeded event with amount, currency and payment identity already checked. Catch only a duplicate violation on the receipt key outside the transaction and acknowledge it. Other failures must remain retryable. Handle out-of-order events; never blindly regress state.

PROVE IT UNDER FAILURE

Send the same signed event concurrently. Confirm one effect. Crash before commit and retry. Use an outbox for emails or other remote effects.

CHECK / Authenticated event IDs are unique and applied atomically.

Framework references: 4, 5 (page 16).

05

Move deferrable work to queues

The user should not wait for CRM synchronization or an email provider. Queue work that can finish later, and make its progress visible.

```
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Queue\Queueable;

class SyncCustomer implements ShouldQueue
{
    use Queueable;
    public int $tries = 3;
    public int $timeout = 20;

    public function __construct(public int $customerId) {}

    public function backoff(): array { return [10, 30]; }

    public function handle(\App\Services\CrmClient $crm): void
    {
        $crm->upsert($this->customerId);
    }
}
```

IMPLEMENTATION NOTES

CrmClient is your application service: use stable identity, explicit HTTP timeouts and throw on failures. Configure an asynchronous queue, supervise workers and alert on failed jobs. Keep retry_after (or visibility timeout) safely above job timeout; for this example, 60 seconds.

PROVE IT UNDER FAILURE

Stop the CRM and verify jobs retry, then become inspectable failures. Replay after recovery without duplicate customers.

CHECK / Background jobs have bounded attempts and recovery ownership.

Framework references: 2 (page 16).

06

Open a circuit during outages

Repeated calls to a broken dependency waste capacity. A circuit rejects calls temporarily and allows a controlled recovery probe.

```
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\Http;
use Illuminate\Http\Client\ConnectionException;

// Minimal one-failure cooldown; see limits below.
abort_if(Cache::has('catalog:open'), 503);
try {
    $r = Http::connectTimeout(1)->timeout(2)
        ->get(config('services.catalog.url').'/items');
} catch (ConnectionException $e) {
    Cache::put('catalog:open', true, 30);
    throw $e;
}
if ($r->serverError()) {
    Cache::put('catalog:open', true, 30);
}
$r->throw();
```

IMPLEMENTATION NOTES

This teaching fragment is a cooldown, not a complete production breaker. Use shared state, atomic failure counters and a half-open probe lock in the real implementation. Specify cache-failure behavior. Opening on every 4xx would hide business errors.

PROVE IT UNDER FAILURE

Verify calls are skipped while open. At cooldown expiry, load-test that only the allowed probe reaches the provider and recovery resets the circuit.

CHECK / A failing dependency can be skipped, then probed safely.

Framework references: 1, 3 (page 16).

07

Isolate dependency capacity

Separate queues protect critical work from slow bulk jobs. Separate worker pools and resource budgets make that isolation meaningful.

```
use App\Jobs\SyncCustomer;
use App\Jobs\SendReceipt;

SyncCustomer::dispatch($customer->id)->onQueue('crm');
SendReceipt::dispatch($order->id)->onQueue('receipts');

// Run under separate supervised process groups:
// php artisan queue:work redis --queue=crm --timeout=20
// php artisan queue:work redis --queue=receipts --timeout=20
```

IMPLEMENTATION NOTES

SendReceipt is an application job. Allocate fixed worker counts to each pool, with independent CPU/memory limits where needed. A single worker consuming both queues does not provide isolation. Also cap concurrency and rate per provider/account.

PROVE IT UNDER FAILURE

Block every CRM worker. Receipts should still be processed within their latency target. Monitor queue age, not only queue length.

CHECK / One busy integration cannot consume every worker.

Framework references: 2 (page 16).

08

Serve an honest fallback

A catalog can show a recent snapshot. A checkout must not claim a payment succeeded because the gateway is unavailable. Pick fallbacks by business meaning.

```
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\Http;
use Illuminate\Http\Client\ConnectionException;
use Illuminate\Http\Client\RequestException;

try {
    $data = Http::connectTimeout(1)->timeout(2)
        ->get(config('services.catalog.url').'/items')
        ->throw()->json();
    Cache::put('catalog:last-good', $data, 600);
    $stale = false;
} catch (ConnectionException | RequestException $e) {
    $data = Cache::get('catalog:last-good');
    abort_if($data === null, 503);
    $stale = true;
}
return response()->json(['data' => $data, 'stale' => $stale]);
```

IMPLEMENTATION NOTES

Validate the payload before caching (pattern 9). This public-catalog example keeps data for at most ten minutes. Scope cache keys by tenant, permissions, currency and query when relevant. Alert on persistent failures even while fallback works.

PROVE IT UNDER FAILURE

Test healthy response, outage with cache, and outage without cache. Confirm stale data is labelled and sensitive data never crosses tenants.

CHECK / Read-only features can degrade without inventing success.

Framework references: 1, 3 (page 16).

09

Validate external contracts

An undocumented field change should become a controlled integration error, not silently corrupt your domain data.

```
use Illuminate\Support\Facades\Validator;

// $response is an HTTP response with status checked.
$payload = $response->json();
$check = Validator::make(
    is_array($payload) ? $payload : [], [
        'id' => ['required', 'string'],
        'amount_minor' => ['required', 'integer', 'min:0'],
        'currency' => ['required', 'in:BRL,USD'],
    ]
);
if ($check->fails()) {
    throw new \UnexpectedValueException(
        'Provider payment contract changed.'
    );
}
$paymentData = $check->validated();
```

IMPLEMENTATION NOTES

Map validated fields into your own DTO or domain model. Match the actual provider schema; validate nested fields, units and enums you depend on. Ignore harmless added fields. Catch this integration exception at your boundary and return a controlled response.

PROVE IT UNDER FAILURE

Remove amount_minor, change its type, or send HTML with status 200. Ensure no payment state changes and an actionable alert is emitted.

CHECK / Unexpected payloads stop at the integration boundary.

Framework references: 6 (page 16).

10

Persist intent with an outbox

A process can crash between saving an order and dispatching its integration job. Write both the order and the delivery record in the same database transaction.

```
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Str;

DB::transaction(function () use ($orderId) {
    DB::table('orders')->where('id', $orderId)
        ->update(['status' => 'confirmed']);

    DB::table('outbox')->insert([
        'id' => (string) Str::uuid(),
        'event_key' => "order-confirmed:{$orderId}",
        'type' => 'order.confirmed',
        'payload' => json_encode(['order_id' => $orderId]),
        'created_at' => now(),
    ]);
});
```

IMPLEMENTATION NOTES

Create an outbox table on the same database connection, with a unique event_key. A relay must claim rows safely, retry delivery and mark sent only after acknowledgement. Delivery is at least once: consumers deduplicate by event ID. Use versioned keys for repeatable transitions.

PROVE IT UNDER FAILURE

Kill the app after commit but before delivery. The relay must recover the row. Kill it after remote success but before marking sent: replay must not duplicate effects.

CHECK / A committed business change cannot lose its delivery intent.

Framework references: 4, 5 (page 16).

11

Make failures observable

Support needs to know which operation failed and whether it can be retried. Measure latency, error rate, queue age and business outcomes per dependency.

```
use Illuminate\Support\Facades\Http;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;

$id = (string) Str::uuid();
$start = hrtime(true);
$status = null;
try {
    $r = Http::withHeaders(['X-Request-ID' => $id])
        ->connectTimeout(1)->timeout(2)
        ->get(config('services.catalog.url').'/items');
    $status = $r->status();
    $r->throw();
} finally {
    Log::info('integration.call', [
        'provider' => 'catalog', 'request_id' => $id,
        'status' => $status,
        'duration_ms' => (hrtime(true) - $start) / 1e6,
    ]);
}
```

IMPLEMENTATION NOTES

Propagate an existing trusted correlation ID across jobs when available. A null status here means no HTTP response was received. Add exception category in your boundary handler. Never log tokens, full payment payloads or unnecessary personal data.

PROVE IT UNDER FAILURE

Force a timeout and a 503. Confirm logs distinguish both, correlate to the user operation, and trigger useful alerts without exposing secrets.

CHECK / Each integration call records safe, correlated evidence.

Framework references: 7 (page 16).

12

Test the unhappy paths

A successful HTTP fake proves little about recovery. Write tests for unknown outcomes, duplicated events, stale reads and exhausted retry budgets.

```
use Illuminate\Support\Facades\Http;

// PHPUnit method in a Laravel Tests\TestCase subclass.
public function test_catalog_outage_is_controlled(): void
{
    Http::preventStrayRequests();
    Http::fake([
        'catalog.test/*' => Http::response([], 503),
    ]);
    config(['services.catalog.url' =>
        'https://catalog.test']);

    // Your route must implement the boundary handling.
    $this->getJson('/api/catalog')
        ->assertStatus(503)
        ->assertJsonPath('message',
            'Catalog temporarily unavailable.');
}
```

IMPLEMENTATION NOTES

This is a test specification for your app, not a supplied route implementation. Add Http::failedConnection() for network errors. Fakes do not measure socket timeouts: use a slow stub for that. Run concurrency tests for locks and unique constraints.

PROVE IT UNDER FAILURE

Cover 429, 5xx, malformed JSON, timeout, duplicate webhook, outbox replay, expired cache and worker restart. Assert final business effects, not only status codes.

CHECK / Failure behavior is verified before a provider outage.

Framework references: 1 (page 16).

One failure. One clear plan.

Use these patterns together. For a payment timeout, the goal is to preserve a truthful state and recover safely.

1 / BOUND THE WAIT

Apply a short request budget so the gateway does not occupy a worker indefinitely.

2 / PRESERVE THE UNKNOWN OUTCOME

Keep the payment pending, with its stable reference and idempotency key. Tell the customer that confirmation is in progress.

3 / RECONCILE SAFELY

Check provider status or process a verified webhook. Replay only according to the provider idempotency contract.

4 / RECOVER THE FOLLOW-UP WORK

Commit confirmed state and outbox intent together. Let isolated workers deliver receipts and downstream updates.

5 / PROVE THE RESULT

Test a lost response after the provider charges. Expect one charge, one confirmed order and a recoverable receipt.

Choose your next improvement

Highest-risk dependency: _____

Failure to simulate: _____

Expected user experience: _____

Owner and deadline: _____

Build software people can trust.

More practical articles and projects:
marcelonizato.dev

Marcelo Nizato | Resilient Developers

Use this guide responsibly.

Examples target Laravel 12.x APIs and PHP 8.2+ syntax. They are focused teaching fragments, not a complete application. Framework calls were checked against the official references below; the snippets have not been executed in a Laravel project.

BEFORE COPYING A SNIPPET

Add it to the appropriate controller, service, job or test. Configure services.* URLs, authentication, database migrations and shared infrastructure. Models, tables, job classes and services named in comments belong to your application. Tune every timeout and retry budget to your workload.

GUARANTEES COME FROM THE WHOLE SYSTEM

Combine application behavior with provider contracts, database constraints and operations. A queue does not guarantee exactly-once effects. A timeout does not prove a payment failed. A fallback must never fabricate business success.

Official Laravel documentation

1. HTTP client: requests, retries and fakes
laravel.com/docs/12.x/http-client

2. Queues: jobs, workers and timeout configuration
laravel.com/docs/12.x/queues

3. Cache: storage and atomic locks
laravel.com/docs/12.x/cache

4. Database: transactions
laravel.com/docs/12.x/database

5. Migrations: indexes and constraints
laravel.com/docs/12.x/migrations

6. Validation: payload rules
laravel.com/docs/12.x/validation

7. Logging: contextual information
laravel.com/docs/12.x/logging

Pattern explanations, scenarios and examples are original educational material. References document the underlying Laravel primitives; outbox and circuit-breaker orchestration require application-specific implementation.