Robust REST APIs

Compatibility, the Request/Action/Resource triad, abilities, pagination, errors, limits

TL;DR

An API is a **contract at the application boundary**: stable names, predictable pagination, useful errors, safe retries. Laravel provides the parts — production quality comes from a deliberate compatibility, authorization and failure policy.

1. Compatibility policy first

```
Route::prefix('v1')->middleware('auth:sanctum')->group(function (): void {
    Route::apiResource('products', ProductController::class);
});
```

Version a **public** API when independently deployed clients need a clear promise. This is **not** a ritual for a private Inertia backend deployed with the server. After v1 ships, don't silently change a field's meaning — publish v2 alongside, set a migration period. Route model binding + a policy decide access; don't duplicate ownership checks in controllers.

2. Separate input / use case / representation

```
// Resource = the only place for public representation
final class ProductResource extends JsonResource {
    /** @return array<string, int|string> */
    public function toArray(Request $request): array {
        return [
            'id' => $this->resource->getKey(),
            'name' => $this->resource->name,
            'price_cents' => $this->resource->price_cents,
        ];
    }
}
// return ProductResource::collection($products) — not an Eloquent collection
```

A Form Request validates input, an action performs the use case, a Resource owns the representation. An accidental model column doesn't become part of the API forever.

3. Narrow token abilities

```
$token = $user->createToken('warehouse-sync', ['products:read']);

Route::get('/v1/products', ProductController::class)
    ->middleware(['auth:sanctum', 'abilities:products:read']);
```

Sanctum tokens are **credentials, not roles**. A policy = the user's access, abilities = the scope of a specific token. A reporting token shouldn't delete products. Rotate, revoke, record the purpose; never a long-lived token in browser JS.

4. Pagination matched to the query

	Offset `paginate()`	**Cursor** `cursorPaginate()`
Gives	total, page numbers	seeks over a stable, indexed order
Use when	a human actually needs page 42	feeds, large append-over-time tables
Risk	inserts shift items; a large offset is expensive	the order must be deterministic (e.g. `created_at, id`)

```
Product::query()->orderBy('id')->cursorPaginate(perPage: 50);   // don't paginate by a computed value
```

5. Machine-readable errors (RFC 9457)

```
// application/problem+json — one recognizable shape
{"type":"https://api.example.com/problems/validation",
 "title":"The request is invalid.","status":422,
 "errors":{"name":["The name field is required."]}}
```

Stable `type`, short `title`, `status`, `instance`/correlation ID; field errors **only** on validation. Centralize the mapping in the exception configuration — controllers return success resources, not assorted error arrays. **Never** send exception messages, SQL or a stack trace to the client.

6. Rate-limit the protected capability

```
RateLimiter::for('partner-api', fn (Request $r): Limit =>
    Limit::perMinute(120)->by((string) $r->user()?->getAuthIdentifier()));
// routes: ->middleware('throttle:partner-api')
```

Named limiters in the provider (the key and policy visible in review). For anonymous traffic — a trusted identity, **not** an untrusted forwarded IP. Return `429` with retry information; alert on a partner being throttled continuously.

Pitfalls / when NOT to

- Versioned REST just "because it sounds modern" — an internal form needs no public promise.
- A cursor over arbitrary sorting without an index and a tie-breaker.
- Sanctum abilities as a full permission model.
- **HTTP retry ≠ exactly-once** — payments and webhooks need an idempotency key + a unique database constraint.
- Before shipping: a feature test of every contract element — success, expected error, authorization, pagination, retry.