Laravel Nexus

Multi-Channel Inventory Sync Driver

An Open-Source Laravel Package for Headless Inventory Synchronization

Document Version	1.0 — Initial Architecture Specification
Tech Stack	Laravel 12, Redis, MySQL, Pest PHP
Supported Channels	Shopify, WooCommerce, Amazon SP-API, Etsy
License	MIT (Open Source)

1. Executive Summary

The multi-channel retail landscape has matured to the point where operating across a single storefront is commercially indefensible. Modern merchants simultaneously list products on Shopify for direct-to-consumer sales, WooCommerce for a WordPress-powered secondary storefront, Amazon for marketplace reach, and Etsy for handcrafted or niche audiences. Each channel maintains its own independent inventory state. When a product sells on Amazon, the stock count on Shopify must be decremented in seconds—not minutes—or the business faces overselling, customer refunds, and account suspension.

The Laravel ecosystem, while providing world-class tooling for web application development, lacks a standardized, open-source solution for this synchronization problem. Developers are currently forced into two unacceptable positions: purchasing expensive SaaS middleware platforms (Cin7, Linnworks, Zoho Inventory) at $200–$500 per month, or writing brittle, channel-specific integration code that cannot scale beyond its original target.

This document proposes Laravel Nexus, a free, open-source Laravel package that provides a unified, driver-based API for synchronizing products, variants, and inventory levels across any combination of supported e-commerce channels. It is not a platform or a hosted service—it is a drop-in package that any Laravel application can adopt via Composer.

For a Senior PHP/Laravel Developer seeking to demonstrate architectural mastery, Laravel Nexus represents the ideal portfolio centerpiece within the e-commerce domain. It showcases the Adapter pattern, advanced queue architecture with job batching, polymorphic data modeling, webhook signature verification, and distributed rate limiting—all core competencies for Senior Architect roles in commerce engineering.

2. The Problem Definition and Market Gap

2.1 The Inventory Consistency Problem

Inventory management across multiple sales channels is fundamentally a distributed systems problem. Every channel maintains its own authoritative stock count. When a sale occurs on any channel, all other channels must be updated atomically—or as close to atomically as their respective APIs permit. The failure modes are severe and commercially damaging:

- Overselling: A product with 1 unit remaining sells simultaneously on Shopify and Amazon before either channel can be updated. The merchant is now committed to two orders for a single item.
- Phantom Stock: A manual count correction on one channel is never propagated, leading to long-term inventory drift that compounds across months.
- Rate Limit Cascades: During peak traffic (e.g., a Flash Sale), a naive synchronizer may attempt to fire hundreds of API requests simultaneously, triggering a 429 response and a temporary ban from the channel's API, creating a synchronization blackout at the worst possible moment.
- Webhook Replay Failures: Channels notify your application of sales events via webhooks. If your server is temporarily unavailable, the webhook is lost. Without a replay mechanism, the inventory event is silently dropped.

2.2 Why Existing Solutions Are Insufficient

The current landscape offers no satisfactory open-source answer. The following table maps the competitive landscape:

Solution	Category	Cost	Key Limitation
Cin7 / Linnworks	SaaS Platform	$299–$499/mo	Hosted, no code-level control. Cannot customize sync logic or add channels.
Zoho Inventory	SaaS Platform	$79–$299/mo	Proprietary API, vendor lock-in. No Laravel integration layer.
Aimeos	Open-Source Platform	Free + dev cost	Full e-commerce platform, not a package. Requires adopting its entire architecture.
Channel-specific Packages	Libraries	Free	Siloed. Each manages one channel. No unified DTO, no cross-channel orchestration.
Custom Spaghetti Code	DIY	Dev hours	Not reusable, brittle, no standardized webhook verification or rate limiting.

The Gap: There Is No 'Horizon for Multi-Channel Inventory'

- Laravel Horizon transformed queue management by providing a unified UI and standardized patterns. Laravel Nexus aims to do precisely the same for inventory synchronization—providing a unified, driver-based abstraction that any developer can install, configure, and extend without rebuilding the core synchronization logic from scratch.

2.3 The Developer's Dilemma

A senior Laravel developer hired to build an e-commerce backend faces a choice with no good options. They can spend 6–12 weeks writing bespoke Shopify, Amazon, and WooCommerce clients—each with its own authentication scheme, rate limiting logic, and webhook signature verification—or pay a monthly SaaS fee that compounds indefinitely. Nexus eliminates this dilemma by providing a production-grade, extensible foundation that a developer can adopt on day one.

3. Project Vision: Laravel Nexus

3.1 Mission Statement

To provide a unified, driver-based, open-source package that any Laravel application can use to synchronize products, variants, and inventory levels across multiple e-commerce channels—without coupling application code to any specific channel's API contract.

3.2 Core Pillars

1. Unification: Normalize every channel's product and inventory model into a single NexusProduct DTO, eliminating the need for channel-specific transformation logic in application code.
2. Reliability: Handle the inherent unreliability of external APIs through job batching, retry logic, and a dedicated Dead Letter Queue for failed sync events.
3. Compliance: Enforce per-channel rate limits using a distributed Token Bucket algorithm backed by Redis, ensuring the package never triggers API bans regardless of how many queue workers are running.
4. Extensibility: Expose a clean driver contract so teams can add support for new channels (eBay, TikTok Shop, Faire) without modifying the package's core.
5. Observability: Provide a Livewire-powered dashboard that visualizes sync health, pending jobs, failed syncs, and a webhook event log for every connected channel.

3.3 Developer Profile Alignment

For a Senior Developer targeting Architect-level roles in e-commerce engineering, this project demonstrates a precise set of high-signal competencies:

- Design Patterns in Action: The Adapter Pattern is demonstrated concretely, not theoretically, by normalizing ShopifyProduct, WooCommerceProduct, and AmazonCatalogItem into a single NexusProduct DTO.
- Advanced Queue Architecture: Syncing 10,000 SKUs across four channels requires Bus::batch(), job chaining, and failure callbacks—not simple dispatch() calls.
- Distributed Systems Thinking: The Token Bucket rate limiter must be race-condition-free across multiple queue workers, requiring atomic Redis operations via Lua scripting.
- Security Engineering: Receiving webhooks from four different providers requires verifying four different HMAC signature schemes, a non-trivial security implementation.
- Data Modeling Sophistication: The polymorphic channel mapping table (nexus_channel_mappings) is a textbook example of when polymorphic relationships solve a real schema problem elegantly.

---

4. Detailed Architecture Specification

4.1 The Driver Contract

The architectural core of Laravel Nexus is the InventoryDriver interface. Every channel—Shopify, WooCommerce, Amazon, Etsy—must implement this contract. Application code interacts only with this interface, remaining fully decoupled from any channel-specific SDK or API structure.

```php
<?php

namespace LaravelNexus\Contracts;

use LaravelNexus\DTOs\NexusProduct;
use LaravelNexus\DTOs\NexusInventoryUpdate;
use Illuminate\Support\Collection;

interface InventoryDriver
{
    public function fetchProduct(string $remoteId): NexusProduct;
    public function pushInventory(NexusInventoryUpdate $update): bool;
    public function listProducts(int $page = 1, int $perPage = 250): Collection;
    public function verifyWebhookSignature(Request $request): bool;
    public function parseWebhookPayload(Request $request): NexusInventoryUpdate;
    public function getRateLimitConfig(): RateLimitConfig;
}
```

4.2 The NexusProduct DTO

The NexusProduct Data Transfer Object is the package's canonical representation of a product. It abstracts away the structural differences between channels. A Shopify product has variants with inventory_item_id. A WooCommerce product has stock_quantity at the root level. An Amazon listing has ASIN and FNSKU. The DTO erases these differences:

```php
<?php

namespace LaravelNexus\DTOs;

use Illuminate\Support\Collection;

class NexusProduct
{
    public function __construct(
        public readonly string  $localSku,
        public readonly string  $title,
        public readonly int     $quantityAvailable,
        public readonly ?string $barcode,
        public readonly Collection $variants,  // Collection<NexusVariant>
        public readonly array   $channelMeta = []  // raw channel-specific payload
    ) {}

    public static function fromShopify(array $payload): self { /* ... */ }
    public static function fromWooCommerce(array $payload): self { /* ... */ }
    public static function fromAmazon(array $payload): self { /* ... */ }
```

```
    public static function fromEtsy(array $payload): self { /* ... */ }
}
```

The channelMeta field preserves the raw channel payload, ensuring that channel-specific operations (e.g., Amazon FBA location routing) are never permanently discarded. This is a deliberate architectural decision: normalize what you need, preserve what you don't.

4.3 The Channel Mapping Table (Polymorphic Relationships)

The most elegant database design challenge in this project is: given a local Product model with a local SKU, how do you store its remote ID on Shopify (a string like 'gid://shopify/Product/789'), its ASIN on Amazon, and its listing_id on Etsy—all without creating three nullable foreign key columns on the products table?

The answer is a dedicated polymorphic channel mapping table:

```
Schema::create('nexus_channel_mappings', function (Blueprint $table) {
    $table->id();
    $table->morphs('syncable');       // syncable_type, syncable_id (your local
Product)
    $table->string('channel');         // 'shopify', 'woocommerce', 'amazon',
'etsy'
    $table->string('remote_id');       // the channel's native ID for this product
    $table->string('remote_sku')->nullable();
    $table->json('channel_meta')->nullable(); // etag, version_id, etc.
    $table->timestamp('last_synced_at')->nullable();
    $table->timestamps();

    $table->unique(['syncable_type', 'syncable_id', 'channel']); // one mapping
per channel
    $table->index(['channel', 'remote_id']); // fast lookup on incoming webhooks
});
```

This design means any Eloquent model in the host application can be made syncable by simply adding the Syncable trait—no schema modifications required to the host application's existing tables:

```
use LaravelNexus\Traits\Syncable;

class Product extends Model
{
    use Syncable; // Adds: $product->channelMappings(), $product-
>syncTo('shopify')
}
```

4.4 The Facade and Developer API

Following Laravel conventions, the primary developer-facing interface is a Facade. The API is designed to be immediately readable and require zero boilerplate for common operations:

```
use LaravelNexus\Facades\Nexus;

// Push a single inventory update to all connected channels
Nexus::product($product)->updateInventory(quantity: 42)->sync();

// Sync to specific channels only
Nexus::product($product)
     ->updateInventory(quantity: 42)
     ->sync(['shopify', 'amazon']);

// Bulk sync an entire catalog (returns a BatchId for monitoring)
$batchId = Nexus::catalog($products)->syncAll();

// Query sync status
$status = Nexus::batch($batchId)->status();
```

5. Distributed Rate Limiting Architecture

5.1 The Problem with Naive Rate Limiting

Each channel imposes strict rate limits that apply at the API key level, not the server level. This is the critical insight that naive implementations miss: if your application runs 4 queue workers and each worker dispatches requests at 2/second, the channel's API sees 8 requests/second against your single API key. The limit is breached, the key is throttled or banned, and all four workers fail simultaneously.

A correct solution must enforce rate limits globally across all workers using a shared state store. Redis is the ideal backing store because it supports atomic operations via Lua scripting, which can decrement a token bucket counter and read its value in a single uninterruptible operation.

5.2 The Token Bucket Algorithm via Redis Lua Script

The Token Bucket algorithm allows for short bursts while enforcing a sustained rate limit. Each channel has its own bucket configuration. The following Lua script is the core of Nexus's rate limiter:

```
-- Token Bucket Lua Script (atomic, race-condition-free)
local key = KEYS[1]
local capacity = tonumber(ARGV[1])   -- max tokens (burst capacity)
local refill_rate = tonumber(ARGV[2]) -- tokens per second
local now = tonumber(ARGV[3])         -- current timestamp (microseconds)
local requested = tonumber(ARGV[4])   -- tokens to consume (usually 1)

local bucket = redis.call('HMGET', key, 'tokens', 'last_refill')
local tokens = tonumber(bucket[1]) or capacity
local last_refill = tonumber(bucket[2]) or now

-- Calculate how many tokens to add since last check
local elapsed = (now - last_refill) / 1000000
tokens = math.min(capacity, tokens + (elapsed * refill_rate))

if tokens >= requested then
    tokens = tokens - requested
    redis.call('HMSET', key, 'tokens', tokens, 'last_refill', now)
    redis.call('EXPIRE', key, 3600)
    return 1  -- request allowed
else
    return 0  -- request throttled
end
```

5.3 Per-Channel Rate Limit Configuration

Each channel has a different rate limit profile, reflecting their respective API tiers. Nexus ships with sensible defaults but allows full override via configuration:

Channel	Default Limit	Burst Capacity	Notes
Shopify	2 req/sec	40 tokens	Leaky bucket in Shopify's API; adjust per plan tier
WooCommerce	25 req/sec	100 tokens	Self-hosted; limit is server-dependent, configurable
Amazon SP-API	1 req/sec	10 tokens	Strict; inventory updates have separate throttle groups
Etsy	10 req/sec	50 tokens	Daily quota applies; Nexus tracks daily request count

6. Queue Architecture and Job Batching

6.1 The Bulk Sync Problem

Syncing a catalog of 10,000 SKUs across 4 channels creates 40,000 individual API calls. These cannot be dispatched naively. The architecture must answer three questions: How are jobs grouped to respect channel rate limits? How is progress tracked across 40,000 disparate jobs? What happens when a batch job fails partway through?

6.2 The Three-Layer Job Architecture

Nexus implements a three-layer job hierarchy that separates orchestration from execution:

Layer 1: CatalogSyncJob (Orchestrator)

- Receives the full collection of products and the target channels.
- Partitions products into batches of 250 (respecting channel page sizes).
- Creates a Bus::batch() containing ChannelSyncBatchJob instances for each partition.
- Registers a then() callback to fire an InventorySyncCompleted event.
- Registers a catch() callback to trigger the Dead Letter Queue handler.

Layer 2: ChannelSyncBatchJob (Channel Executor)

- Receives a chunk of 250 products and a single target channel.
- Before each API call, acquires a rate limit token from Redis (blocking with exponential backoff if throttled).
- Dispatches individual PushInventoryJob instances for any products that fail at this layer.
- Reports progress using $this->batch()->incrementProcessed().

Layer 3: PushInventoryJob (Atomic Unit)

- Handles a single product-channel pair.
- Implements ShouldBeUnique to prevent duplicate jobs for the same SKU-channel combination.
- On failure, writes a detailed failure record to nexus_sync_failures for dashboard display.
- On success, updates the last_synced_at timestamp on the channel mapping record.

6.3 The Dead Letter Queue

Jobs that exhaust all retry attempts are not silently discarded. They are written to a nexus_dead_letter_queue table with the full payload, the exception message, and the number of attempts made. The Livewire dashboard surfaces these failures with a 'Replay' button, allowing developers to re-dispatch a failed job after the root cause has been resolved—without needing artisan access or custom scripts.

```
Schema::create('nexus_dead_letter_queue', function (Blueprint $table) {
    $table->id();
    $table->string('channel');
    $table->string('job_class');
    $table->json('payload');            // full job payload for replay
    $table->text('last_exception');
    $table->integer('attempts');
    $table->string('status')->default('failed'); // failed | replaying | resolved
    $table->timestamp('failed_at');
    $table->timestamps();
});
```

7. Unified Webhook Receiver

7.1 The Signature Verification Problem

Receiving webhooks securely is more complex than most tutorials demonstrate. Each channel uses a different HMAC signature scheme:

Channel	Header	Algorithm	Key Source
Shopify	X-Shopify-Hmac-Sha256	HMAC-SHA256 (base64)	Webhook secret from Partner Dashboard
WooCommerce	X-WC-Webhook-Signature	HMAC-SHA256 (base64)	Secret set on WC webhook object
Amazon	X-AMZ-SNS-Signature	RSA-SHA1 (AWS SNS cert)	Public cert from SNS endpoint URL
Etsy	X-Etsy-Signature	HMAC-SHA256 (hex)	API shared secret from Developer Console

Amazon's SNS-based webhook verification is particularly complex—it requires downloading a public certificate from a URL embedded in the request payload, verifying the certificate itself against Amazon's trusted domains, and then verifying the signature. Nexus handles all of this transparently.

7.2 The Unified Webhook Route

Nexus registers a single webhook endpoint that accepts events from all channels via a channel discriminator in the URL:

```
// In NexusServiceProvider::boot()
Route::prefix('nexus/webhooks')->group(function () {
    Route::post('{channel}', NexusWebhookController::class)
         ->name('nexus.webhook')
         ->middleware(VerifyNexusWebhookSignature::class);
});

// URLs registered with each channel:
// POST https://your-app.com/nexus/webhooks/shopify
// POST https://your-app.com/nexus/webhooks/woocommerce
// POST https://your-app.com/nexus/webhooks/amazon
// POST https://your-app.com/nexus/webhooks/etsy
```

7.3 Standardized Event Dispatching

Once a webhook payload is verified and parsed into a NexusInventoryUpdate DTO, Nexus fires a standardized Laravel Event. Application code listens to this single event type regardless of which channel triggered it:

```
// Dispatched after every successful webhook—from any channel
event(new InventoryUpdated(
    channel: 'shopify',
    product: NexusProduct::fromShopify($payload),
    previousQuantity: $previousQuantity,
    newQuantity: $update->quantity,
));

// Application code listens to ONE event, not four channel-specific events:
class PropagateInventoryToOtherChannels
{
    public function handle(InventoryUpdated $event): void
    {
        // Shopify just sold 1 unit. Update Amazon, WooCommerce, and Etsy.
        $channels = ['amazon', 'woocommerce', 'etsy'];
        Nexus::product($event->product)
             ->updateInventory($event->newQuantity)
             ->sync($channels);
    }
}
```

8. The Livewire Dashboard

8.1 Overview

The Nexus dashboard is a Livewire 3 Single Page Application mounted at /nexus by the service provider. It requires no additional frontend build tooling—it ships as part of the package with compiled assets. The dashboard transforms what would otherwise be opaque background job activity into a real-time, actionable control surface.

8.2 Dashboard Components

8.2.1 Channel Status Cards

Four cards (one per channel) display a real-time health indicator: Connected (green), Throttled (amber), or Disconnected (red). Each card shows the current token bucket fill level, the last successful sync timestamp, and the number of jobs currently in the queue for that channel. This component polls every 3 seconds via wire:poll.3s.

8.2.2 Sync Job Monitor

A real-time table showing all active Bus::batch() instances. Each row shows the batch ID, the triggering event, total jobs, processed jobs, failed jobs, and an animated progress bar. Clicking a batch row expands a drawer showing individual job statuses.

8.2.3 Webhook Event Log

A paginated, searchable log of every webhook received in the past 30 days. Each row shows the channel, event type (order.created, inventory.update), the triggering remote ID, verification status

(PASSED / FAILED), and the processing duration. Failed verification attempts are highlighted in red and are searchable to support security auditing.

8.2.4 Dead Letter Queue Manager

A table of all jobs in the nexus_dead_letter_queue with their failure reasons. Each row has a Replay button that re-dispatches the job with its original payload and a Resolve button that marks the failure as acknowledged without replaying. A bulk replay action allows recovering from an API outage where dozens of jobs may have failed simultaneously.

8.3 Real-Time Alerts via Laravel Reverb

For teams using Laravel Reverb, Nexus optionally broadcasts circuit-level alerts in real time. When an Amazon sync job encounters its 5th consecutive 429 response (configurable), Nexus fires a ChannelThrottled event that implements ShouldBroadcast. The dashboard immediately renders a banner: 'ALERT: Amazon SP-API is throttling requests. 14 jobs are queued.' This demonstrates mastery of Laravel's complete real-time event stack.

9. Extensibility: Adding Custom Channels

9.1 The Driver Registration Model

A core design goal is that the package's core never needs to be modified to support a new channel. Any developer can implement the InventoryDriver interface, register it, and Nexus will treat it identically to a first-party channel. This is the Adapter Pattern in its most commercially valuable form.

```
// In AppServiceProvider::boot()
Nexus::extend('ebay', function ($app) {
    return new EbayInventoryDriver(
        config: $app['config']['nexus.channels.ebay'],
        client: $app->make(EbayApiClient::class)
    );
});

// Now 'ebay' is a first-class citizen:
Nexus::product($product)->updateInventory(10)->sync(['shopify', 'ebay']);
```

This extensibility model means the package's value grows with the ecosystem. Community-contributed drivers for TikTok Shop, Faire, Walmart Marketplace, and others can be published as separate Composer packages without any changes to the Nexus core.

9.2 Event Hooks for Application-Level Customization

Beyond driver extensibility, Nexus exposes event hooks at every critical junction in the sync pipeline. Application developers can listen to these events to implement custom business logic:

- BeforeInventorySync: Fired before each push. Can be used to apply business rules (e.g., 'never sync inventory below safety stock level of 5 units').
- AfterInventorySync: Fired on success. Can trigger downstream workflows like notifying a warehouse management system.
- InventorySyncFailed: Fired on permanent failure (after retries). Can trigger an alert to a Slack channel or PagerDuty.
- WebhookReceived: Fired before processing. Can be used to log all inbound webhooks to a separate audit table.

10. Complete Database Schema

The following tables are published to the host application via php artisan vendor:publish --tag=nexus-migrations:

Table	Primary Purpose	Key Columns

Table	Primary Purpose	Key Columns
nexus_channel_mappings	Links local models to remote IDs	syncable_type, syncable_id, channel, remote_id, last_synced_at
nexus_sync_jobs	Active job tracking	batch_id, channel, status, payload, attempts, created_at
nexus_dead_letter_queue	Permanently failed jobs	channel, payload, last_exception, attempts, status
nexus_webhook_logs	Inbound webhook audit log	channel, event_type, remote_id, verified, processed_at
nexus_rate_limit_logs	Rate limit event tracking	channel, tokens_remaining, throttled, requested_at

11. Testing Strategy

11.1 Test Architecture

Laravel Nexus follows a test pyramid structure with Pest PHP as the testing framework. The absence of a robust test suite would undermine the project's claim to production-readiness, so testing is treated as a first-class deliverable.

11.1.1 Unit Tests: DTO Normalization

Every fromChannel() factory method on NexusProduct is tested with fixture payloads captured from each channel's actual API responses. These tests verify that the normalization logic is correct and that no data is silently dropped.

11.1.2 Integration Tests: Driver Contracts

Each driver is tested against a mocked HTTP client (using Laravel's Http::fake()) with pre-recorded responses. These tests verify the full request-response cycle: authentication headers, correct endpoint construction, and correct parsing of both success and error responses.

11.1.3 Concurrency Tests: Rate Limiter

The Token Bucket rate limiter is tested against a real Redis instance (using a test database) with concurrent requests dispatched via parallel PHP processes. These tests verify that the Lua script is truly atomic and that no race conditions allow requests to exceed the configured rate limit.

11.1.4 Feature Tests: Webhook Verification

Each channel's webhook verification is tested using real HMAC signatures computed from known payloads. Tests also verify that tampered payloads (signature mismatch, replayed timestamps) are correctly rejected with a 401 response.

12. Implementation Roadmap

Phase 1: Foundation (Weeks 1–2)

Goal: A working command-line inventory sync between a Laravel application and Shopify.

- Set up package scaffolding using spatie/package-skeleton-laravel.
- Define and implement the InventoryDriver interface and NexusProduct DTO.
- Build the ShopifyDriver with OAuth 2.0 authentication and REST API integration.
- Implement the nexus_channel_mappings migration and the Syncable trait.
- Write comprehensive Pest PHP tests for the Shopify driver with Http::fake() fixtures.

Phase 2: Multi-Channel Expansion (Weeks 3–4)

Goal: Full support for all four channels.

- Implement WooCommerce driver (Basic Auth, REST API v3).
- Implement Amazon SP-API driver (LWA token refresh, HMAC-SHA256 request signing).
- Implement Etsy driver (OAuth 2.0 PKCE flow, keysets API).
- Build the distributed Token Bucket rate limiter with Lua scripting.
- Write concurrency tests for the rate limiter.

Phase 3: Queue Architecture (Week 5)

Goal: Production-grade bulk sync with job batching.

- Implement the three-layer job hierarchy (CatalogSyncJob, ChannelSyncBatchJob, PushInventoryJob).
- Build the Dead Letter Queue handler and the nexus_dead_letter_queue table.
- Implement the ShouldBeUnique constraint on PushInventoryJob.

Phase 4: Webhooks (Week 6)

Goal: Secure, unified webhook receiving.

- Implement the VerifyNexusWebhookSignature middleware with per-channel strategies.
- Build the Amazon SNS certificate verification flow.
- Implement the nexus_webhook_logs table and the WebhookReceived event.

Phase 5: The Dashboard (Weeks 7–8)

Goal: Full Livewire dashboard with real-time monitoring.

- Implement the four Channel Status Card components.

- Build the Sync Job Monitor with Bus::batch() progress reporting.
- Build the Webhook Event Log with search and pagination.
- Build the Dead Letter Queue Manager with Replay and Resolve actions.
- Optional: Add Laravel Reverb integration for real-time alerts.

Phase 6: Documentation and Launch (Week 9)

Goal: Portfolio-ready, publicly publishable package.

- Write comprehensive documentation covering installation, configuration, driver creation, and security considerations.
- Record a demo video showing a real inventory sync from a Laravel application to Shopify and Amazon simultaneously.
- Publish to Packagist and submit to Laravel News.
- Write an architectural deep-dive blog post covering the Adapter Pattern, Token Bucket rate limiting, and polymorphic channel mapping—targeting the developer audience that will most appreciate the architecture.

---

13. Competitive Positioning and Strategic Value

13.1 Vs. SaaS Platforms (Cin7, Linnworks)

SaaS platforms offer breadth (many integrations) but sacrifice depth and control. Nexus offers the developer complete ownership: the sync logic lives in their codebase, can be version-controlled, customized, and extended without API access or subscription fees. For agencies building bespoke commerce backends, this is a decisive advantage.

13.2 Vs. Manual Integration

A developer building channel integrations from scratch faces months of work: OAuth flows, API client libraries, rate limit handling, webhook verification, error handling, retry logic, and monitoring. Nexus reduces this to a Composer install and a configuration file. The developer retains full control of the sync logic through events and hooks, without writing the infrastructure from scratch.

13.3 The 'Senior Architect' Signal

Most portfolio projects demonstrate that a developer can build features. This project demonstrates that a developer can build infrastructure that other developers build features on top of. That is the critical distinction between a Senior Developer and a Senior Architect. Nexus sends this signal unambiguously by implementing:

- A published, versioned Composer package with a stable public API.
- A driver extension system that third-party developers can contribute to.
- A security implementation (webhook HMAC verification, including Amazon's RSA-based SNS scheme) that demonstrates awareness of attack surfaces.

- A distributed, race-condition-free rate limiter that demonstrates understanding of concurrency in multi-process PHP environments.
- A test suite covering unit, integration, and concurrency scenarios.

---

14. Conclusion

Laravel Nexus occupies a genuine gap in the Laravel ecosystem. No open-source package currently provides a unified, driver-based, production-ready solution for multi-channel inventory synchronization. The SaaS alternatives are expensive and inflexible. The DIY approach is fragile and non-reusable.

This project is architecturally rich enough to serve as a portfolio centerpiece for a Senior PHP/Laravel Developer. It demonstrates the Adapter Pattern, advanced queue architecture, distributed rate limiting, polymorphic data modeling, webhook security, and operational tooling—all within the domain of e-commerce, which represents a large fraction of real-world Laravel applications.

More importantly, it solves a problem that every developer working in multi-channel commerce has encountered. A package that solves a real problem with architectural elegance is the clearest possible demonstration that a developer has crossed the threshold from “senior engineer” to “systems architect.”

---

composer require vendor/laravel-nexus | Sync Everywhere. Build Once.

End of Document — Laravel Nexus Project Specification v1.0