HYPERCE -- ENGINEERING BLOG

Vendure in the Wild

Six real stories from building production e-commerce on Vendure.

TOPICS

Job Queues -- B2B Pricing -- Search & Indexing Multi-Channel -- Flutter Integration -- Framework Overview

BY

The Hyperce Team

CONTENTS

At Hyperce, we build production Vendure storefronts every day. These posts are honest field notes from real projects -- the problems we hit, the decisions we made, and what we learned. No theory. No slides. Just the stuff that actually matters when you go live.

POST 01

The Day My Vendure Worker Decided to Send Every Customer Two Emails

Dev Diary -- Job Queues -- 5 min read

Three weeks after going live, a client messaged me: customers were getting duplicate order confirmations. Some were getting none at all.

It was a Tuesday. I had just made coffee.

The culprit? Multiple worker processes with no safeguards around job execution, all polling the same job queue. Under load, two workers could pick up the same job almost simultaneously. Both would run it. Double email. Classic.

Vendure handles background work through a job queue. Your main server handles API requests, while a separate worker processes tasks like emails, inventory updates, and order state changes.

In development with a single worker, everything looks flawless. In production with several workers running in parallel, you need to think about concurrency.

> **In development with a single worker, everything looks flawless. In production with several workers, you need to think about concurrency.**

The fix was not complicated. For simple deployments, making jobs idempotent solves most of it -- check whether the action has already happened before performing it again.

For more serious setups, switching the default SQL-backed queue to the Redis-based queue plugin using BullMQ adds proper distributed locking and more reliable job handling across multiple workers.

One afternoon of debugging. One configuration change. Problem gone.

Vendure is genuinely one of the best foundations you can build e-commerce on: TypeScript-first, GraphQL API, plugin architecture, and an admin dashboard included. It handles the complex commerce logic so you can focus on building something people actually want to use.

You just have to understand the moving pieces before they surprise you in production.

> **That is exactly what the team at Hyperce has already done -- production-ready Vendure with the sharp edges handled, so you can skip the Tuesday afternoon debugging sessions entirely.**

#jobqueue · #vendure · #workers · #concurrency · #bullmq

POST 02

We Needed B2B Pricing. Vendure's Plugin System Made Me Feel Like a Wizard.

Plugin Architecture -- B2B Commerce -- 5 min read

The client sold industrial components. Sounds boring until you learn they had six tiers of customer pricing, region-specific discounts, volume breaks, and a rule that trade account holders never see retail prices -- ever.

Before we started, my instinct was to reach for a monolith platform with every bell and switch built in. That is usually how these projects end up as a mess of config screens nobody understands six months later.

We went with Vendure instead, and specifically we leaned into the plugin system harder than I expected we would need to.

Vendure exposes a PriceCalculationStrategy interface. You implement it, register it, and the platform routes every price calculation through your code. No monkey-patching. No overriding internals. Just a clean extension point.

We wrote a CustomerGroupPricePlugin. It reads the customer group at request time, pulls the right price tier from a separate table we own, and returns the adjusted figure. The rest of Vendure -- promotions, tax, the admin UI -- does not know or care. It just sees a price.

> **No monkey-patching. No overriding internals. Just a clean extension point -- and it just worked.**

The first time it worked end-to-end I sat back and stared at the screen. The trade account customer saw their negotiated price. The guest saw retail. No leakage, no hacks.

What impressed me more was how little we had to touch to add a new pricing rule three months later when the client added a seasonal discount tier. Open the plugin, add a condition, redeploy. Done.

That is the promise of a well-designed plugin architecture. Not just that you can extend things, but that extensions do not collapse into each other over time.

Getting there still takes some architecture thinking up front. You have to understand how Vendure's extension points fit together before you start building against them.

> **Hyperce has done that groundwork. Built on Vendure in production B2B environments, the patterns we have developed mean you are not figuring out the plugin architecture from scratch when a client asks for something unusual at 4pm on a Friday.**

#b2bcommerce · #plugins · #vendure · #pricingtiers · #typescript

POST 03

The Search Box That Broke My Confidence -- and How Vendure Put It Back Together

Search & Indexing -- Performance -- 6 min read

It started with a Slack message from a QA tester: searching for 'waterproof jacket' returns no results but the product is definitely in there. I told her to hard-refresh. That was not it.

The store had about 14,000 SKUs. Not massive by enterprise standards, but enough that Vendure's default SQL-backed search plugin was starting to feel the weight. Faceted filtering was slow. Partial matches were not matching. And the relevance ranking had the personality of a coin flip.

Vendure's default search works fine for getting off the ground. It is SQL full-text search under the hood, and for a catalogue in the hundreds it is perfectly adequate. At thousands of products with complex facets, you start to feel friction.

The platform anticipated this. There is an official Elasticsearch plugin, and the migration path is more graceful than I expected. You swap the search plugin in your configuration, point it at your Elasticsearch instance, and trigger a reindex. Your GraphQL search queries do not change. Your storefront does not change. The interface is the same -- only the engine behind it is different.

The reindex took about forty minutes for our catalogue. Then we ran the same query.

Instant. Relevant. Typo-tolerant. The QA tester sent a thumbs up emoji and went back to her checklist.

> **Your GraphQL queries do not change. Your storefront does not change. Only the engine behind it is different.**

What struck me most was how little the rest of the system cared. Vendure's layered architecture meant we replaced a significant piece of infrastructure and nothing outside the search module noticed.

That is not something you get for free. It is the result of someone making deliberate decisions about where abstractions should sit. Vendure has those abstractions in the right places.

The thing is, knowing which plugin to reach for and when -- that institutional knowledge takes time to build. You collect it through projects like this one, through the moment you realise the default tool is not the right tool anymore.

> **Hyperce has already collected it. We have run Vendure at scale, navigated these exact inflection points -- when to upgrade, what to swap, how to do it without breaking the storefront. That accumulated judgment is what you are actually getting when you work with a team that lives in this ecosystem.**

#search · #elasticsearch · #vendure · #performance · #indexing

POST 04

One Catalogue, Four Countries, Zero Separate Codebases

Multi-Channel -- Internationalisation -- 5 min read

The brief seemed reasonable at first glance: a fashion brand, four European markets, different pricing and currencies per country, some products exclusive to certain regions, and a single team managing everything from one admin dashboard.

I have seen this requirement kill projects. The usual failure mode is one Frankenstein installation held together by environment variables and prayers, or four separate Vendure instances that slowly drift out of sync as different developers touch different things.

Vendure has a concept called Channels. Each Channel is a lens onto your catalogue -- it has its own currency, tax settings, price list, and assigned products. You can have a Channel per country, per storefront, per B2B segment. One database. One admin. Multiple views.

Setting this up is not a five-minute job. You have to think clearly about what is shared versus what is channel-specific. Get that wrong and you will spend a lot of time untangling things.

We spent two days on an architecture document before writing a single line of code. Where do promotions live -- global or per channel? What happens when a product goes out of stock in Germany but not Spain? How does the admin team assign something to France without accidentally touching the UK store?

> **One database. One admin. Four storefronts -- each showing its own prices, its own currency, its own product selection.**

Once you have answers to those questions, Vendure's Channel system delivers on the promise. We built a single Next.js storefront that reads the active channel from a subdomain, passes the channel token to the Vendure GraphQL API on every request, and receives the right prices, right currency, right product list automatically.

The brand's content team logs into one admin panel. They assign a product to channels, set a channel-specific price, and it appears on the correct storefronts. No dev involvement. No deployment.

Three months after launch the client added a fifth market -- Switzerland, CHF, some products excluded. Configuration change. No code changes to the storefront. Live in a day.

Multi-channel Vendure is powerful. But the value is only there if you architect it correctly from the start.

> **At Hyperce we have run multi-channel, multi-region Vendure stores. The architecture decisions that look obvious in hindsight -- we have already made them, argued about them, and found the configuration that actually holds up when a fifth country gets added on short notice.**

#multichannel · #i18n · #vendure · #channels · #ecommerce

POST 05

My Experience Integrating Vendure with Flutter to Build an E-Commerce App

Mobile -- Flutter -- GraphQL -- 4 min read

As a Flutter developer, I often focus on building clean, responsive mobile interfaces. Recently, I worked on an e-commerce project where I needed a powerful backend that could handle products, orders, customers, and checkout logic -- without building it all from scratch.

Vendure stood out immediately. It is a headless commerce platform, which means the backend focuses purely on business logic and APIs while the frontend can be built with any technology. In my case, Flutter handled the entire user interface.

Why Vendure for Mobile

A few things made Vendure the right fit: the GraphQL API works beautifully with mobile apps, the ready-to-use admin dashboard meant I could manage products and orders without building a separate admin panel, and the TypeScript-first architecture made it easy to extend when needed.

The architecture was clean and simple. The Flutter app communicates with Vendure through GraphQL, Vendure processes the request and talks to the database. Flutter App -- GraphQL Client -- Vendure Shop API -- PostgreSQL. That separation meant the mobile app and backend could be developed independently.

State Management with BLoC

To manage application state I used the BLoC pattern. BLoC kept the UI separate from the business logic, making the project easier to maintain and test. The structure was layered cleanly: UI events fire into the Bloc, which calls a repository, which calls the GraphQL service, which hits the Vendure API.

Once the GraphQL integration was working, retrieving products, updating carts, and managing orders became straightforward. Vendure manages carts through something called an Active Order -- a user creates it, products get added, quantities updated, and the user proceeds to checkout. Each step is a GraphQL mutation Vendure provides out of the box.

Challenges Along the Way

Vendure uses a structured order lifecycle during checkout. At first, understanding the different states an order goes through before completion took some time. Some GraphQL operations also

return different response types depending on success or failure, and handling these cases correctly in Flutter required careful parsing.

Managing authentication sessions -- maintaining the session token across requests and ensuring authenticated calls stayed authenticated -- also needed deliberate handling of headers.

> **Flutter made it easy to build a smooth mobile interface, while Vendure handled the complex e-commerce backend.**

Integrating Vendure with Flutter turned out to be a great combination. For developers building a mobile e-commerce application, this stack offers a strong balance between flexibility and powerful backend capabilities. If you need a robust backend for a Flutter commerce app, Vendure is worth exploring.

> **At Hyperce, we have taken this stack to production. We know the integration patterns, the BLoC structures that hold up at scale, and the Vendure configuration that makes mobile commerce feel effortless to the end user.**

#flutter · #vendure · #graphql · #mobilecommerce · #bloc

POST 06

Crafting an E-Commerce App Has Never Been This Easy Using Vendure

Framework Deep-Dive -- Developer Experience -- 3 min read

Building a production-ready and scalable e-commerce application has always been a prime concern for software developers. When starting a new project, developers face several challenges: choosing the right architecture, selecting a reliable tech stack, and ensuring the system can scale as the business grows.

Ever since I started working with Vendure, I can confidently say that almost half of these hassles disappeared. Many of the common problems involved in building an e-commerce backend are already handled by Vendure right from the bootstrap stage.

What Makes Vendure Different

Vendure is a developer-first headless commerce framework designed to build flexible and scalable e-commerce backends. Unlike traditional platforms, Vendure strictly follows the Separation of Concerns principle. This allows developers to build storefronts using any technology stack while relying on Vendure to handle backend commerce logic.

Vendure is built on top of NestJS, which means it inherits a clean and modular architecture. Modules are implemented as plugins, making it easy to extend functionality without modifying the core system.

Why Not Build From Scratch?

A common question: why not design the system from scratch? While that may provide full control, it also means solving problems that mature frameworks have already solved -- and solved well.

After working with Vendure for nearly a year, I can say it provides more than enough flexibility for real-world platforms. Even when client requirements do not match Vendure's default behaviour, customizing or intercepting the framework is not difficult. The community is strong, and the core maintainers are genuinely active in helping users resolve issues.

The Features That Matter Most

TypeScript-first. Vendure is built entirely in TypeScript -- strong typing, excellent IDE autocompletion, safer refactoring, and better maintainability for large codebases.

Plugin-based architecture. You extend Vendure without modifying the core. Custom business logic, new GraphQL APIs, external integrations, background jobs, custom admin UI features -- all through plugins.

GraphQL flexibility. Fetch only the data you need, extend the schema with custom queries, maintain strongly typed API contracts. It makes building modern frontends much cleaner.

Custom Fields. One of the most powerful features. Extend existing entities -- Products, Variants, Orders, Customers -- without modifying the core schema. Adapt Vendure to specific business requirements while maintaining full framework compatibility.

> **After nearly a year with Vendure, it has become one of my favourite tools. If you are looking for a flexible, production-ready commerce framework -- this is it.**

Building scalable e-commerce platforms traditionally requires significant engineering effort. Vendure solves many of these problems by providing a developer-friendly, extensible, and scalable foundation for modern commerce systems.

> **Hyperce was built on exactly this conviction. We chose Vendure because it gives us the control and extensibility real clients need, without the overhead of reinventing what the framework already handles beautifully. Every project we ship is proof of it.**

#vendure · #typescript · #graphql · #nestjs · #ecommerce · #plugins

hyperce.io Vendure Engineering Blog