</> FOR DEVELOPERS

Build a Django E-Commerce Platform

Catalog, cart, Stripe checkout, orders — a complete online store.

BUILT WITH

- Django
- Python
- PostgreSQL
- Stripe
- Redis
- HTMX

Catalog & Variants | Cart & Checkout | Stripe Payments | Orders & Inventory

TECHNOVIZE PUBLISHING

</> djangozen.com

TECHNOVIZE PUBLISHING

Build a Django E-Commerce Platform from Scratch

Catalog, cart, Stripe checkout, orders, and everything a real online store needs

KC Ramo

Build a Django E-Commerce Platform from Scratch
Catalog, cart, Stripe checkout, orders, and everything a real online store needs

First Edition, 2026

Published by Technovize Publishing
Amsterdam, The Netherlands

Cover and interior design by Technovize Publishing.

Contents

Reference & Checklists

Preface

An online store looks simple from the outside — a grid of products, a cart, a checkout button — and is deceptively demanding underneath. It handles money, which must be exactly right; it tracks stock, which two shoppers can race for at the same instant; it takes payments through an external processor whose confirmations arrive asynchronously and out of order; and every second of slowness or every bug is a lost sale. The gap between a tutorial cart and a store you can actually take orders through is wide, and it is made of exactly these unglamorous concerns. This book closes that gap by building one real e-commerce platform, end to end, on Django.

Who should read this book

This book is for Django developers who want to build a genuine online store. If you are comfortable with Django — models, views, templates, the ORM, migrations — you have the foundation you need. You do not need prior experience with Stripe, inventory systems, or the finer points of pricing and tax; each is introduced where the build requires it. What you will gain is the ability to build a store that takes payments correctly, never oversells, keeps customers' data and orders safe, and stays fast enough to convert — the difference between a shopping-cart demo and software a business runs on.

What you will build

Throughout the book we build **ShopFlow**, a complete e-commerce platform: customers browse a catalog of products and variants, add items to a cart, check out and pay with Stripe, and receive orders that are tracked through fulfillment — with inventory, discounts, customer accounts, an operations back-office, and everything a real store needs. It is deliberately a full store rather than a toy, because the hard parts of e-commerce are not any single feature but the way money, stock, payments, and orders interact. Learn to build ShopFlow well and you can build a store for any product.

How the book is organized

The book follows the order you would actually build in, grouped into five parts. **Part I, Foundations**, sets up the project and models the catalog of products and variants. **Part**

II, The Storefront, builds the customer-facing store: listing and detail pages, the shopping cart, and search and discovery. **Part III, Checkout & Payments**, is the money path: the checkout flow, taking payments with Stripe, and the order lifecycle. **Part IV, Running the Store**, adds inventory management, discounts and pricing, and customer accounts. **Part V, Production**, covers the admin and operations, security and performance, deployment and scaling, and launch and growth. Each chapter ends with a summary and key takeaways, and the appendices gather the checklists and references you will return to.

Conventions and two non-negotiables

Code appears in monospaced blocks, shown where it lives, with imports trimmed once a pattern is established. Filenames are given relative to the project root. The recurring principle is that **production-ready means being honest about trade-offs** — where a decision matters, the book weighs the alternatives. Two things in e-commerce are not trade-offs but absolutes, and they run through the whole book. First, **money must be exact**: prices are stored and computed as integer minor units (cents), never floating-point, and totals are always recomputed on the server. Second, **stock and payment must be correct under concurrency**: two customers buying the last unit, or a payment webhook arriving twice, must never corrupt your data. Much of what separates this book's store from a demo is the care taken with exactly these two things.

A note on Stripe and change

We take payments with Stripe because it is the pragmatic, well-documented choice, and because it lets card data bypass your servers entirely — a huge reduction in risk and compliance burden. Stripe's API details will evolve after this book is written, but the patterns here — server-side amounts, webhooks as the source of truth, idempotency, and never trusting the browser for payment confirmation — are durable and apply to any payment provider. Learn the patterns, and the specifics will follow. With that, let us begin by getting clear on what we are building and how an online store fits together.

PART I

Foundations

CHAPTER 1

Introduction

Most Django tutorials teach you to build a shopping cart in an afternoon. You add a `Product` model, wire up a session-based cart, drop a Stripe button on a page, and call it done. Then you try to take real money from a real stranger over the internet, and the whole thing falls apart. The customer's card gets charged but the order never saves. Two people buy the last unit in stock. A discount code that should have expired keeps working. A webhook arrives twice and you ship two packages. The tutorial cart was never wrong, exactly. It was just never asked to be right when it mattered.

This book is about the gap between those two things. We are going to build **ShopFlow**, a complete e-commerce platform, from an empty directory to a system you could actually run a small store on. Not a demo. Not a toy. A store that takes money correctly, tracks stock correctly, survives duplicate webhooks, handles the failure cases, and gives you and your staff the tools to operate it day to day. Along the way, the recurring theme of this book is a single uncomfortable idea: **production-ready means being honest about trade-offs.** There is no configuration flag that makes software correct. There are only decisions, made with your eyes open, about what can fail and what you will do about it.

What ShopFlow Is, and What It Will Do

ShopFlow is an online store. By the end of this book, a customer will be able to land on the storefront, browse a catalog organized into categories, search and filter products, and open a product page with images, variants, price, and stock status. They will add items to a cart that persists whether or not they are logged in, adjust quantities, apply a discount code, and see shipping and tax reflected in the total. They will check out, pay with a real card through Stripe, and receive a confirmation email with an order number. Behind that, an order will be recorded exactly once, stock will be decremented exactly once, and the payment will be reconciled against Stripe's own record of what happened.

On the other side of the same application, you — the operator — will have a staff-facing area where you can see incoming orders, mark them as paid, packed, and shipped, issue refunds, manage the catalog, adjust inventory, and create discount campaigns. Background jobs will send emails, sync inventory, expire abandoned carts, and retry work that failed the first time. The whole thing will run in Docker behind Nginx and Gunicorn,

with PostgreSQL for durable data, Redis for caching and queues, and Celery for the background work.

Here is the shape of what a customer's core journey touches, end to end:

```
Browse catalog  →  Product page  →  Add to cart  →  Cart review
      →  Checkout (address, shipping)  →  Payment (Stripe)
      →  Order confirmed  →  Confirmation email  →  Fulfillment
```

Every arrow in that diagram is a place where something can go wrong, and most of this book lives on the arrows, not the boxes. The boxes — the models and the pages — are the easy part. The arrows — the transitions, the money changing hands, the stock moving — are where correctness is won or lost.

The Anatomy of an E-Commerce Platform

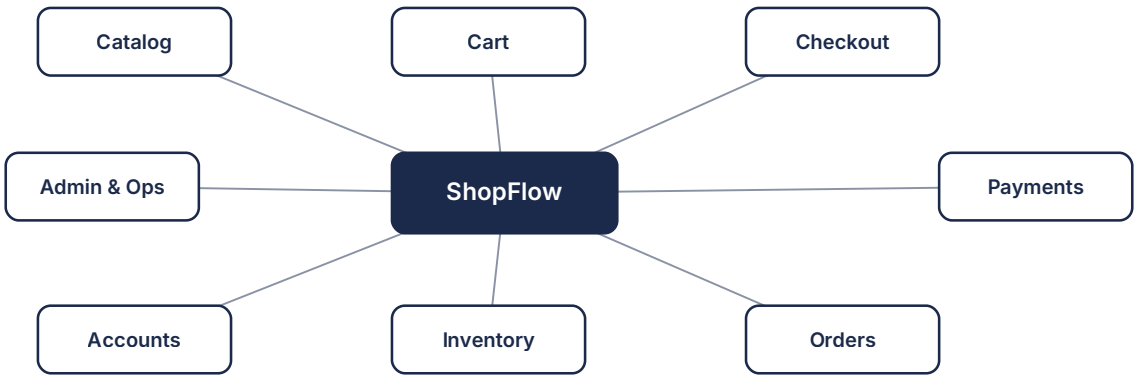

Figure 1.1 — The anatomy of ShopFlow: a dozen small subsystems that have to agree with each other.

Before we write a line of code, it helps to name the parts. An e-commerce platform is not one system; it is a dozen small systems that have to agree with each other. Understanding these pieces up front tells you why the later chapters are organized the way they are.

- **Catalog.** The products you sell: their names, descriptions, images, prices, categories, and variants (size, colour, format). This is the part everyone gets right, because it is mostly reading and displaying data. It is also where search, filtering, and merchandising live.
- **Cart.** The temporary, mutable collection of what a shopper intends to buy. Carts are surprisingly subtle: they must work for anonymous visitors, survive login, merge when

an anonymous user signs in, and reflect price and stock changes without letting a stale price sneak through to checkout.

- **Checkout.** The funnel that turns a cart into an order: collecting the shipping address, choosing a shipping method, calculating tax, and preparing a payment. Checkout is a conversion-critical, high-anxiety moment for the customer and a high-risk moment for your data.
- **Payments.** Actually moving money. We use Stripe, and we treat Stripe — not our own database — as the source of truth for whether a payment succeeded. This distinction drives a large part of the architecture.
- **Orders.** The permanent record of a completed sale. An order is immutable in the ways that matter (you do not get to quietly change what the customer paid) and mutable in the ways that don't (its fulfillment status moves from paid to shipped). Orders are the backbone the rest of operations hangs on.
- **Inventory.** How much of each thing you have. Stock is the second piece of state, after money, that must be correct. Overselling is not a rounding error; it is a promise you cannot keep and a refund you have to issue.
- **Discounts.** Coupon codes, percentage-off, fixed-amount, free shipping, first-order, minimum-spend. Discounts are a small feature with an enormous surface area for bugs, because they interact with price, tax, stock, and usage limits all at once.
- **Accounts.** Customer identity: registration, login, saved addresses, order history. Also the guest path, because forcing account creation before checkout is one of the most reliable ways to lose a sale.
- **Admin and staff tools.** The interface your team uses to run the store. Django gives us a strong head start here, but the built-in admin is not a fulfillment console, and we will be deliberate about where it stops being enough.
- **Operations.** Everything that keeps the system alive: background jobs, emails, logging, monitoring, backups, and deployment. This is invisible to customers and indispensable to you.
- **Growth.** The concerns that appear once the store works at all: performance under load, SEO, analytics, and the ability to change the system without breaking it.

Notice how much of this list has nothing to do with displaying products. A store is mostly the plumbing between the shelf and the till. That is exactly the plumbing tutorials skip.

Why Build It on Django

You could build a store on almost any stack. We choose Django deliberately, and it is worth being explicit about why, because the reasons shape how we use it.

Django is a **batteries-included** framework, and e-commerce needs a lot of batteries. Authentication, sessions, forms, an ORM with real transactions, a migration system, a templating engine, CSRF protection, and a usable admin all ship in the box and all work together. When you are building something with as many moving parts as a store, the value of a framework where the parts already fit is hard to overstate. You spend your effort on the store, not on gluing a router to a template engine to a database library.

Django's ORM gives us **transactions and row-level locking** as first-class tools. When we decrement stock or record a payment, we need database guarantees, not hopeful application code. Django exposes `transaction.atomic()`, `select_for_update()`, and constraints directly, and PostgreSQL underneath makes them real. A large fraction of "correctness of money and stock" comes down to using these primitives well, and Django makes them accessible without hiding what they do.

Django is **server-rendered by default**, and for a store that is a feature, not a limitation. Product pages that render on the server are fast to first paint, cheap to cache, and friendly to search engines — all of which matter directly to revenue. We add interactivity with HTMX and Alpine.js where it earns its keep, rather than shipping a single-page application and a separate API you have to keep in sync. Fewer moving parts, fewer places to be wrong.

Finally, Django is **boring in the best way.** It is mature, widely deployed, well documented, and unlikely to reinvent itself out from under you between chapters. For software that handles other people's money, boring is a virtue. You want a foundation whose failure modes are known.

The Tech Stack, and Why Each Piece

Every dependency is a liability you accept in exchange for leverage. Here is the full stack and the honest reason each piece is in it.

Component	Choice & why
Django 5.2 (Python 3.12+)	The application framework and our home base. We target the current stable Django and a modern Python because async views, better typing, and performance improvements are real, and because building a new project on an old runtime is starting in debt.
PostgreSQL	The durable store of record. We choose Postgres over SQLite (which cannot handle concurrent writes the way a store needs) and over MySQL for its stronger transactional semantics, rich constraint support, `SELECT ... FOR UPDATE` behaviour, and features like partial indexes and `JSONB` that we will actually use. Money and stock live here.
Redis	An in-memory data store we use for two jobs: caching (rendered fragments, expensive queries, session data) and as the message broker for Celery. Redis is fast, simple, and exactly right for work that is allowed to be lost on restart. Crucially, we never make Redis the source of truth for anything we cannot afford to lose.
Celery	The background task queue. Sending an email, syncing inventory to a warehouse, generating an invoice PDF, retrying a failed webhook — none of these should happen inside the request that a waiting customer is staring at. Celery moves that work off the request path so the customer gets a fast response and the slow work happens reliably in the background, with retries.
Stripe	The payment processor. We do not touch raw card numbers; Stripe does, and takes the compliance burden with it. We use Stripe Payment Intents and, critically, Stripe webhooks as the authoritative signal that money moved. An entire chapter is devoted to doing this correctly, because this is where tutorials are most dangerously wrong.
HTMX + Alpine.js	Progressive interactivity on top of server-rendered HTML. HTMX lets us swap page fragments (add-to-cart, quantity updates, filtering) without full reloads; Alpine handles small client-side state (dropdowns, modals). Together they give a modern feel without a build pipeline, a client-side router, or a duplicated API. This is a deliberate trade-off: we accept a slightly less "app-like" ceiling in exchange for radically less complexity.
Docker, Gunicorn, Nginx	The deployment surface. Docker makes the environment reproducible; Gunicorn runs the Python application; Nginx terminates TLS, serves static files, and sits in front as the reverse proxy. This is a conventional, well-understood production topology, and conventional is what you want for infrastructure.

You will notice what is *not* here: no React front-end framework, no GraphQL layer, no microservices, no Kubernetes. Those are not wrong in general. They are wrong for a single-team store at this scale, where they would add operational cost far exceeding the benefit. Choosing the smaller stack is itself an engineering decision, and the honest kind — we are optimizing for a team that has to ship and then live with what they shipped.

The Business Realities That Shape the Engineering

It is tempting to treat an e-commerce platform as a pure engineering problem, but the constraints that make it hard come from the business, and you cannot make good technical decisions without holding them in mind.

Every bug is lost money or a lost customer. In most software, a bug is an annoyance you fix in the next release. In a store, a bug on the checkout page during a sale is revenue that evaporates and never comes back. A customer who hits an error at the payment step usually does not retry; they leave, and they leave with a bad impression. This changes the risk calculus for every feature that touches the funnel. The checkout flow does not get "we'll fix it later" treatment. It gets built carefully the first time.

Correctness of money is non-negotiable. If the total is wrong by one cent, you have either shortchanged the customer or shortchanged yourself, and at scale both are real problems and one is fraud. This is why we never store money as a floating-point number — `float` cannot represent `0.10` exactly, and the errors accumulate. We use fixed-precision decimals and integer cents, and we compute totals in exactly one place so there is one answer, not three that disagree. Consider the difference:

```
>>> 0.1 + 0.2
0.30000000000000004          # float: silently wrong
>>> from decimal import Decimal
>>> Decimal("0.1") + Decimal("0.2")
Decimal('0.3')               # decimal: exact, auditable
```

That tiny difference is the seed of a real financial bug. A store that computes tax on thousands of orders using floats will, over time, produce numbers that do not reconcile with Stripe's records, and reconciliation is not optional when the tax office asks.

Correctness of stock is non-negotiable. If two customers can buy the same last unit, one of them gets a cancellation email and a refund, and you get a support ticket and a reputation cost. The naive pattern — read the stock count, check that it is greater than zero, then decrement it — contains a race condition that is invisible in development

(where you are the only user) and inevitable in production (where two requests arrive in the same millisecond). Solving it correctly requires database-level locking or atomic conditional updates, and we will do exactly that. Getting this right is one of the load-bearing lessons of the book.

Operations are part of the product. The store is not done when a customer can pay. It is done when your team can find an order, refund it, see why a webhook failed, restore last night's data, and deploy a fix without downtime. A store nobody can operate is not a store; it is a liability with a nice front page. We treat the staff experience and the operational tooling as first-class features, not an afterthought.

What "Production" Means Versus a Tutorial Cart

The word "production-ready" gets thrown around loosely, so let us define it concretely for this book. A tutorial cart and a production store can look identical on the happy path — a logged-in user, buying an in-stock item, with a card that succeeds on the first try. The difference is everything that happens off the happy path. Here is the contrast, feature by feature:

- **Payments.** Tutorial: charge the card, redirect to a success page, trust that it worked. Production: create a Payment Intent, confirm success through a signed webhook, handle the case where the browser closes between charge and redirect, and make the whole flow **idempotent** so a duplicate webhook does not create a duplicate order.
- **Stock.** Tutorial: `product.stock -= 1`. Production: an atomic, locked decrement inside a transaction that refuses to go below zero, so concurrent buyers cannot oversell.
- **Orders.** Tutorial: create the order after redirect. Production: create the order exactly once, tie it to the payment, keep it immutable in the parts that represent the contract with the customer, and record enough history to answer "what happened to order 4021?" a month later.
- **Money.** Tutorial: `price * quantity` with floats. Production: decimals, integer cents, tax and shipping computed in one authoritative place, totals that reconcile with Stripe to the cent.
- **Discounts.** Tutorial: subtract a percentage. Production: enforce usage limits and expiry under concurrency, so a single-use code cannot be redeemed twice by two simultaneous requests.

- **Failure.** Tutorial: assume success. Production: assume anything can fail — the network, Stripe, the email server, the database connection — and decide, for each, whether to retry, roll back, or alert a human.
- **Observability.** Tutorial: `print()`. Production: structured logs, error tracking, and enough signal to diagnose a problem you cannot reproduce.

The pattern across every row is the same: production code takes the failure cases as seriously as the success case, and it treats money and stock as things that must be provably correct, not merely usually correct. That is the standard we hold ShopFlow to. Where a shortcut is tempting, we will name it, explain what it costs, and choose deliberately — sometimes we will even take the shortcut, but never without telling you the price.

Who This Book Is For

This book is written for developers who already know Django well enough to be dangerous and want to learn to be reliable. You should be comfortable with the fundamentals: models, migrations, views, templates, the ORM, and the request/response cycle. You should have built at least one Django project past the tutorial stage. You do not need prior e-commerce experience, and you do not need to be a payments expert — we build that understanding from the ground up.

If you have followed a "build a shop" tutorial before and came away feeling that it skipped all the hard parts, this book is aimed squarely at you. We spend our time exactly where those tutorials go quiet: concurrency, money, idempotency, failure handling, and operations. If you are a solo founder or a small team building a real store, you will finish with a system you can actually run. If you work on a larger platform, you will find the patterns — the service layer, the selectors, the idempotent payment flow, the locked stock decrement — transfer directly to more complex systems.

This book is *not* a Python primer, not an introduction to Django, and not a front-end design course. We use HTMX and Alpine.js pragmatically and keep the styling functional rather than fashionable; you can dress the store up later. Our focus is the engine, not the paint.

How the Book Is Organized

ShopFlow is built in five parts, each layering on the last. The order is deliberate: we build the durable, correctness-critical core before the features that depend on it, so that by the time money changes hands, the ground underneath it is solid.

- **Part I — Foundations.** Project setup, Docker environment, PostgreSQL and Redis, settings organized for multiple environments, and the architectural spine of the whole application: the service layer, selectors, and the rule that views and models stay thin. We establish how we test and how we deploy before we build features, because retrofitting either is painful.
- **Part II — The Catalog.** Products, variants, categories, images, search, and filtering. This is where you get comfortable with the codebase and the patterns while the stakes are low, because nothing here moves money. It is also where we set up caching and think about performance for pages customers hit constantly.
- **Part III — Cart and Checkout.** The cart that survives login and merges correctly, the checkout funnel, addresses, shipping methods, and tax. We compute totals in one authoritative place and confront the price-consistency problem: the price at checkout must be the price the customer agreed to, not whatever the catalog says right now.
- **Part IV — Payments and Orders.** The heart of the book. Stripe Payment Intents, webhooks as the source of truth, idempotent order creation, the atomic locked stock decrement, discounts under concurrency, refunds, and the order lifecycle from paid to shipped. If you read one part twice, read this one.
- **Part V — Operations and Growth.** Staff tooling and fulfillment, Celery background jobs, transactional email, logging and monitoring, backups, performance and load, and deploying safely. This is what turns a working application into a running business.

Each chapter builds real, runnable code on top of the last. You can read straight through and end with a complete store, or treat any part as a reference for its topic. The code is written to be read: we favour clarity over cleverness, and where we make a non-obvious choice, we explain it in prose rather than leaving you to reverse-engineer it.

A Tour of the Finished Store

To orient yourself, here is where we are headed. A visitor arrives at the storefront and sees a home page with featured products and categories. They browse into a category,

filter by attributes and price, and open a product page showing a gallery, variant selectors, live stock status, and an add-to-cart button that updates a cart badge without reloading the page — that is HTMX doing quiet work.

They add a few items, open the cart, adjust a quantity, and paste in a discount code, which validates and adjusts the total in place. They proceed to checkout as a guest, enter a shipping address, pick a shipping method, and watch tax and total recalculate. They reach the payment step, where Stripe's card element is embedded securely, and pay. Their browser lands on a confirmation page with an order number, and within seconds a confirmation email arrives — sent by a Celery worker, not by the request they just finished.

Behind that single click, a great deal happened correctly: a Payment Intent was confirmed, a webhook arrived and was verified, an order was created exactly once, stock was decremented atomically, the discount's usage counter went up under a lock, and a background job was queued for the email. If any of those steps had failed, the system would have known what to do — roll back, retry, or flag for a human.

Meanwhile you, the operator, open the staff area. You see the new order, its payment confirmed and reconciled against Stripe. You mark it packed, then shipped, which triggers a shipping-confirmation email. You glance at inventory, top up a low-stock variant, and create a weekend discount campaign with a usage cap and an expiry date. You check the dashboard and see that yesterday's totals reconcile with Stripe to the cent. Nothing is on fire, because the boring, careful work was done up front.

That is ShopFlow. Not glamorous, but honest and correct — the kind of software you can hand real customers and real money and then sleep at night. Let's build it.

Summary

This chapter framed the whole book. We are building ShopFlow, a complete Django e-commerce platform, from an empty directory to a store you could operate for real — with a catalog, cart, checkout, Stripe payments, orders, inventory, discounts, accounts, staff tooling, and the operational plumbing that keeps it alive. We named the parts of an e-commerce platform and saw how much of it lives in the plumbing between the shelf and the till, not in displaying products. We justified the stack — Django, PostgreSQL, Redis, Celery, Stripe, HTMX and Alpine, Docker/Gunicorn/Nginx — with an honest reason for each piece and an equally honest note about what we deliberately left out. Most

importantly, we established the standard: production means taking the failure cases as seriously as the success case, and treating the correctness of money and stock as non-negotiable. That standard, and the theme that production-ready means being honest about trade-offs, will guide every decision in the chapters ahead.

Key Takeaways

- **The gap between a tutorial cart and a real store is the failure cases.** The happy path looks identical; production is everything that happens when a step goes wrong.
- **Money and stock must be provably correct, not usually correct.** Use decimals and integer cents for money, atomic locked updates for stock, and compute totals in exactly one place.
- **Stripe, not your database, is the source of truth for payments.** Confirm success through verified webhooks, and make order creation idempotent so duplicate events cannot create duplicate orders.
- **Every bug in the funnel is lost revenue or a lost customer,** which changes the risk calculus and means checkout gets built carefully the first time, not fixed later.
- **Django earns its place** through batteries-included maturity, real transactions and row-level locking, server-rendered speed and SEO, and the boring reliability you want around other people's money.
- **The stack is chosen for a small team that has to ship and then live with what it shipped** — deliberately excluding heavier tools whose operational cost would exceed their benefit at this scale.
- **Operations are part of the product.** Staff tooling, background jobs, logging, backups, and safe deployment are first-class features, not afterthoughts.
- **The book builds in five layered parts** — Foundations, Catalog, Cart and Checkout, Payments and Orders, Operations and Growth — establishing the correctness-critical core before the features that depend on it.

End of sample

You have read the opening chapter of

Build a Django E-Commerce Platform from Scratch

The complete book continues from here — every chapter, every appendix, in PDF and EPUB.

EUR 34.00

Get the full book

https://djangozen.com/ebooks/book/build-django-ecommerce-platform/