# WordPress E-Commerce with WooCommerce

## A Complete 10-Chapter Web Platforms Course

**Topics covered:**

WooCommerce as a plugin, not a platform · products, variations & inventory

Store design & template overrides · cart, checkout & the order lifecycle

Tokenized payment gateways & real PCI-DSS scope · shipping & tax, including nexus & VAT

Security for checkout, orders & accounts · hooks & custom functionality

Performance at scale · Core Web Vitals for store pages

**Builds directly on:** WordPress Fundamentals & WordPress Intermediate/Advanced

**Capstone:** a real store, built and hardened end to end

**Exercises:** 30 hands-on scenarios with worked solutions

**Format:** A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

# Table of Contents


01 What WooCommerce Actually Is

02 Products, Variations & Inventory

03 Store Design

04 Cart, Checkout & Order Flow

05 Payment Gateways

06 Shipping & Tax Configuration

07 Security for E-Commerce Sites

08 Extending WooCommerce: Hooks & Custom Functionality

09 Performance at Scale

10 Capstone: Building and Securing a Complete Online Store

CHAPTER 1 OF 10

# What WooCommerce Actually Is

## WordPress E-Commerce with WooCommerce

Chapter 1 · What WooCommerce Actually Is

*WordPress Intermediate/Advanced*'s own capstone closed with an honest scope note: no WooCommerce, no e-commerce — a genuine, deliberate gap left open for exactly this course to fill. This chapter assumes both *WordPress Fundamentals* and *WordPress Intermediate/Advanced* are already familiar territory, and starts from the single idea everything else in this course builds on: WooCommerce is not a separate platform bolted onto WordPress. It's a plugin.

### Still Just a Plugin — Nothing New to Install Around It

Every mechanism *WordPress Fundamentals* already taught for installing and running plugins applies to WooCommerce unchanged: it's found through **Plugins → Add New**, it activates the same way, it lives in the same `wp-content/plugins/` directory, and it can conflict with other plugins or themes exactly the way *WordPress Fundamentals 6*'s own plugin-vetting chapter warned about. There is no separate login, no separate database, no separate hosting requirement.

| ORDINARY WORDPRESS | WORDPRESS + WOOCOMMERCE, ACTIVATED |
|---|---|
| Same `wp-admin` dashboard | The identical dashboard — WooCommerce adds new menu items to it, it doesn't replace it |
| Same theme system | The same active theme still renders every page — WooCommerce supplies templates the theme can override, not a competing theme engine |
| Same user roles from *WordPress Fundamentals 8* | WooCommerce adds a `Customer` and `Shop Manager` role on top of the existing five — the same capability system underneath |
| Same MySQL database | Store data lives in the same database — mostly the same core tables, plus a small number of WooCommerce-specific ones (see below) |

**WHY THIS FRAMING MATTERS FOR THE REST OF THE COURSE**

Because WooCommerce is "just" a plugin running on ordinary WordPress, almost everything this course covers is really *WordPress Intermediate/Advanced*'s own material — custom post types, hooks, the REST API, security hardening — applied to one specific, very well-documented real-world case. This course teaches relatively little that's conceptually brand new; it teaches how a familiar toolkit gets used for a genuinely high-stakes purpose.

## What WooCommerce Actually Adds: A Custom Post Type Called `product`

*WordPress Intermediate/Advanced 4* covered `register_post_type()` as the mechanism behind custom content types. WooCommerce doesn't invent a new concept here — it calls that exact same function, once, to register its own `product` post type, conceptually equivalent to this:

```php
<?php
// conceptually what WooCommerce does on activation — not code you need to write yourself
register_post_type( 'product', array(
    'label'        => 'Products',
    'public'       => true,
    'has_archive'  => 'shop',
    'supports'     => array( 'title', 'editor', 'thumbnail' ),
) );
?>
```

Every product is, underneath, a row in the same `wp_posts` table every page and post already lives in — with `post_type = 'product'` instead of `'post'` or `'page'`. Product-specific data — price, SKU, stock quantity — is stored as post meta in `wp_postmeta`, the identical mechanism *WordPress Intermediate/Advanced 4* already covered for custom fields on any post type.

> **A GENUINE, EVOLVING EXCEPTION WORTH NAMING PRECISELY: ORDER DATA**
>
> Older WooCommerce versions stored every order the same way — as a post, with `post_type = 'shop_order'`. Since WooCommerce 8.2 (2023), stores can opt into **High-Performance Order Storage (HPOS)**, which moves order data out of `wp_posts`/`wp_postmeta` entirely and into dedicated tables (`wp_wc_orders` and related tables), purpose-built for the query patterns a busy store's order history actually needs. As of WooCommerce 8.5+, HPOS is the default for new stores. This is the one deliberate, documented exception to "it's all just posts and postmeta" — worth knowing precisely rather than assumed away, since it directly affects Chapter 4's own order-flow material and any custom code Chapter 8 writes that queries orders directly.

## Installing WooCommerce & the Setup Wizard

Installation itself is unremarkable — **Plugins → Add New → search "WooCommerce" → Install Now → Activate**, exactly the flow *WordPress Fundamentals 6* already covered. Activating it launches a guided setup wizard that this course will return to, piece by piece, rather than rushing through now:

| SETUP WIZARD STEP | COVERED IN DEPTH IN |
|---|---|
| Store details (address, currency, industry) | This chapter — one-time configuration, no further depth needed |
| Product types you plan to sell | Chapter 2 — Products, Variations & Inventory |
| Theme / storefront appearance | Chapter 3 — Store Design |

| SETUP WIZARD STEP | COVERED IN DEPTH IN |
|---|---|
| Payment methods | Chapter 5 — Payment Gateways |
| Shipping & tax | Chapter 6 — Shipping & Tax Configuration |

**THE WIZARD'S DEFAULTS ARE A STARTING POINT, NOT A FINISHED STORE**

Every setting the wizard asks for can be changed later under **WooCommerce → Settings** — nothing chosen during setup is permanent. Treat the wizard as a fast path to a working test store, not a decision that locks anything in.

## What This Course Covers

Products and inventory, store design, the cart/checkout flow, payment gateways, shipping and tax, security specific to e-commerce, extending WooCommerce with hooks, and performance at scale — closing with a capstone that builds and hardens one complete store end to end. Chapter 7's own security chapter is where this course pays off the exact gap *WordPress Intermediate/Advanced*'s capstone named explicitly: nonces, escaping, and `$wpdb->prepare()` applied specifically to checkout and account forms, where the stakes are real money and real customer data rather than a defaced blog post.

## Hands-On Exercises

**EXERCISE 1**

A colleague claims WooCommerce "isn't really WordPress anymore" once it's installed, since the site now looks and behaves so differently. Using this chapter's own material, explain precisely what is and isn't actually different underneath.

View solution

**EXERCISE 2**

Explain where a WooCommerce product's title and description are stored versus where its price and SKU are stored, and name the two underlying WordPress database tables involved, referencing WordPress Intermediate/Advanced 4's own material on custom post types.

View solution

**EXERCISE 3**

Explain what High-Performance Order Storage (HPOS) changes about where order data lives, and why this chapter calls it "the one deliberate, documented exception" to WooCommerce's usual posts/postmeta storage model.

View solution

### CHAPTER 1 QUICK REFERENCE

- WooCommerce is a **plugin**, not a separate platform — same dashboard, same theme system, same role/capability system, same underlying database
- WooCommerce registers a `product` custom post type via the same `register_post_type()` mechanism from *WordPress Intermediate/Advanced 4*
- Product data lives in `wp_posts` (title/description) and `wp_postmeta` (price, SKU, stock) — the same tables any custom post type uses
- **High-Performance Order Storage (HPOS)** — since WooCommerce 8.2, orders can live in dedicated tables instead of `wp_posts`; default for new stores since 8.5+
- Installation is the standard plugin flow from *WordPress Fundamentals 6*; the setup wizard's choices are all editable later under **WooCommerce → Settings**
- **Next chapter:** Products, Variations & Inventory

CHAPTER 2 OF 10

# Products, Variations & Inventory

## WordPress E-Commerce with WooCommerce

Chapter 2 · Products, Variations & Inventory

Chapter 1 established that a product is, underneath, a post of type `product` with its price and stock stored as post meta. This chapter goes one level deeper: WooCommerce doesn't have just one kind of product, and one of those kinds — variable products — quietly relies on a second custom post type this course hasn't named yet. Then it covers the two pieces of data every product type shares: SKUs and stock.

### The Four Product Types

Every product is assigned exactly one **product type**, chosen from a dropdown on the product's edit screen. This single setting changes which fields appear on the rest of the screen and how the product behaves at checkout.

| TYPE | WHAT IT IS | TYPICAL USE |
|---|---|---|
| Simple | One price, one SKU, one stock count — no options to choose | A single physical item with no size/color/etc. choices |
| Variable | A parent product with multiple purchasable **variations**, each with its own price/SKU/stock, driven by attributes like Size or Color | A T-shirt sold in several sizes and colors |
| Grouped | A collection of existing simple products, displayed and purchasable together, with no price or stock of its own | A "starter bundle" page linking to several already-listed products |
| External / Affiliate | No cart or checkout involvement at all — a "Buy product" button links straight to another site | An affiliate listing that sells through Amazon, Etsy, or a partner store |

**ONLY SIMPLE AND VARIABLE PRODUCTS GO THROUGH THIS COURSE'S OWN CHECKOUT**

Chapters 4 and 5 (cart/checkout and payment gateways) are written around Simple and Variable products specifically — the two types that actually enter WooCommerce's own cart. Grouped products route to their underlying Simple products' own add-to-cart buttons; External/Affiliate products never touch WooCommerce's checkout at all.

### Variations: A Second Custom Post Type, Hiding in Plain Sight

A Variable product first defines one or more **attributes** — for example, `Size` with values Small/Medium/Large, and `Color` with values Red/Blue. Each specific combination you choose to sell (Small/Red, Medium/Blue, and so on) becomes its own **variation**, with its own price, SKU, and stock quantity.

> **A PRECISE DETAIL WORTH NAMING, NOT GLOSSING OVER**
>
> Each variation is not just another row of post meta on the parent product — it's a separate post, of type `product_variation`, with `post_parent` pointing back to the Variable product's own post ID. This is the exact same parent/child post relationship pattern WordPress uses elsewhere (a Page's own sub-pages use `post_parent` the same way) — WooCommerce didn't invent a new relationship mechanism for variations, it reused one already built into WordPress core.

Practically, this means a Variable product with 6 size/color combinations is really **7 posts** in `wp_posts`: one `product` post (the parent, holding the shared title, description, and images) and six `product_variation` child posts (each holding its own price, SKU, and stock in its own postmeta).

## SKUs — Your Own Inventory Identifier, Meeting WooCommerce's

A **SKU** (Stock Keeping Unit) is a code you assign yourself — WooCommerce doesn't generate or require any particular format. Its only real constraint is that, if you set one at all, it must be **unique** across every product and variation in the store; WooCommerce will refuse to save a duplicate.

- SKUs are optional at the WooCommerce level, but rarely optional in practice — they're the identifier most shipping carriers, accounting software, and inventory systems expect to integrate against
- A Simple product has one SKU; a Variable product's parent typically has none, since each of its variations carries its own SKU instead
- A sensible SKU scheme (e.g. `TSHIRT-RED-M`) pays off directly once Chapter 6 covers shipping classes and Chapter 9 covers reporting at scale — an unplanned or missing SKU scheme is a real, common source of later confusion in a growing store

## Stock Management

Stock tracking is opt-in per product, under the **Inventory** tab of the product editor:

| SETTING | WHAT IT CONTROLS |
|---|---|
| `Manage stock?` | Turns quantity tracking on for this product; leaving it off treats the product as always available |
| `Stock quantity` | The current count — automatically decremented by WooCommerce as orders are placed |
| `Low stock threshold` | Triggers a notification to the store admin, without changing what customers see |
| `Allow backorders?` | Controls whether a product can still be ordered once its stock quantity reaches zero |

| SETTING | WHAT IT CONTROLS |
| --- | --- |
| `Stock status` | The customer-facing label — In stock / Out of stock / On backorder |

**GROUPED AND EXTERNAL/AFFILIATE PRODUCTS DON'T USE THIS SYSTEM DIRECTLY**

A Grouped product has no stock of its own — availability is entirely a function of its underlying Simple products' own stock. An External/Affiliate product's stock, if tracked at all, is purely informational, since WooCommerce's own checkout is never actually involved in that sale. Chapter 9's own performance material returns to stock queries specifically for Simple and Variable products, where WooCommerce's checkout genuinely depends on an accurate count.

## Hands-On Exercises

**EXERCISE 1**

A store owner wants to sell a hoodie in three sizes and two colors, each combination priced and stocked separately. Which product type should they use, and why would Simple or Grouped not fit this case?

View solution

**EXERCISE 2**

Explain how many posts, and of what post types, exist in wp_posts for a Variable product with 2 attributes and 4 total variations, including the parent. Connect your answer to this chapter's own post_parent explanation.

View solution

**EXERCISE 3**

Explain why an External/Affiliate product's "stock quantity," if set at all, doesn't function the same way a Simple product's stock quantity does.

View solution

**CHAPTER 2 QUICK REFERENCE**

- **Four product types:** Simple, Variable, Grouped, External/Affiliate — only Simple and Variable go through WooCommerce's own checkout
- **Variations** are real posts of type `product_variation`, children of the parent Variable product via `post_parent` — the same WordPress-core parent/child mechanism, not a new one

- **SKUs** are self-assigned, must be unique store-wide, and matter most for shipping/accounting/inventory integration
- **Stock management** is opt-in per product — quantity, low-stock threshold, backorders, and customer-facing status are all independently configurable
- Grouped and External/Affiliate products don't use stock management the same way, since neither routes through a real checkout of its own quantity
- **Next chapter:** Store Design

CHAPTER 3 OF 10

# Store Design

## WordPress E-Commerce with WooCommerce

Chapter 3 · Store Design

Chapters 1 and 2 covered what a product *is* underneath. This chapter covers how it actually gets shown to a shopper — WooCommerce's own theme-override system, the official Storefront theme, page-builder compatibility, and the store-specific shortcodes and blocks used to place shop content anywhere on the site.

### WooCommerce's Own Template Override System

*WordPress Intermediate/Advanced 2* already covered the WordPress template hierarchy — how `single.php`, `page.php`, and similar theme files decide how content renders. WooCommerce extends that same idea with its own parallel hierarchy: every page it renders (Shop, single product, Cart, Checkout, My Account) comes from a template file bundled inside the WooCommerce plugin itself, under `woocommerce/templates/`.

A theme can override any of these by copying the file into an identically-structured `woocommerce/` folder inside the theme, and editing the copy:

```
# the plugin's own original file
wp-content/plugins/woocommerce/templates/content-product.php

# the theme's override — same relative path, inside the theme's own folder
wp-content/themes/your-theme/woocommerce/content-product.php
```

**EVERY OVERRIDE IS A FORK YOU NOW OWN**

Once a template is copied into the theme, WooCommerce stops using its own version for that page and uses the theme's copy instead — permanently, until it's removed. If WooCommerce later updates that same file (a bug fix, a new feature, a security fix), the theme's own frozen copy never receives it automatically. *WordPress Intermediate/Advanced 5*'s plugin-development chapter already introduced actions and filters as a lower-risk alternative for smaller changes — Chapter 8 of this course returns to that exact idea specifically for WooCommerce, since a hook survives an update that a copied template file doesn't.

### Storefront — The Official WooCommerce Theme

**Storefront** is a free theme built and maintained by Automattic (the same company behind WooCommerce itself), designed specifically around WooCommerce's own template structure rather than adapted to it afterward. It isn't the only theme that works with WooCommerce — *any* reasonably modern WordPress theme will render a store in some form — but it's the reference implementation worth understanding first.

| STOREFRONT (WOOCOMMERCE-NATIVE) | A GENERAL-PURPOSE THEME WITH BASIC WOOCOMMERCE SUPPORT |
|---|---|
| Store layout, cart icon, and checkout flow designed around WooCommerce from the start | WooCommerce pages render, but may need real CSS work to match the theme's own design language |
| Maintained by the same team that ships WooCommerce itself — updates track each other closely | Compatibility depends entirely on that theme's own developer keeping pace with WooCommerce's changes |
| A minimal, deliberately unopinionated starting point — expected to be customized further | Often more visually distinctive out of the box, at the cost of a less predictable WooCommerce integration |

## Page Builders & WooCommerce

Popular page builders (Elementor, Beaver Builder, and similar) generally offer dedicated WooCommerce widgets — a product grid, an "Add to Cart" button, a mini-cart — built specifically to read real store data rather than treating a store as static content. Elementor's own paid "Pro" tier, for example, includes a full WooCommerce Builder for customizing the product and archive templates visually, instead of copying and editing PHP template files by hand.

> **A PAGE BUILDER DOESN'T REPLACE THIS CHAPTER'S OWN OVERRIDE SYSTEM — IT WRAPS IT**
>
> Under the hood, a page builder's WooCommerce widgets are still rendering the same underlying template structure this chapter already covered; the builder just provides a visual editor for it instead of requiring direct file edits. The same update-safety trade-off named above still applies to any heavy customization made through a builder.

## Store-Specific Shortcodes & WooCommerce Blocks

Two separate mechanisms exist for placing store content on any ordinary page — a legacy **shortcode** system, and the newer **WooCommerce Blocks** built for the Block Editor *WordPress Fundamentals 4* already covered in depth.

```
<!-- shortcodes: typed directly into a Classic-style paragraph or Shortcode block -->
[products limit="4" columns="4" category="hoodies"]
[add_to_cart id="123"]
[woocommerce_cart]
[woocommerce_checkout]
[woocommerce_my_account]
```

| SHORTCODES (LEGACY) | WOOCOMMERCE BLOCKS (CURRENT) |
|---|---|
| Plain text tags, typed or pasted in | Native Gutenberg blocks — *Products*, *Cart*, *Checkout* — configured through the block editor's own sidebar, matching *WordPress Fundamentals 4*'s own block-editing workflow |
| Still fully supported — most existing stores still use some of them | The direction WooCommerce itself is actively investing in, including a newer block-based checkout |
| Configuration lives entirely in the shortcode's own attribute syntax | Configuration is visual, with live preview inside the editor |

**A REAL, RECENT SHIFT WORTH NAMING PRECISELY**

A block-based Checkout (and Cart) block, built as a genuine alternative to the older `[woocommerce_checkout]` shortcode-driven page, became broadly available starting with WooCommerce 8.3 (late 2023) and has continued to mature since. New stores are increasingly built on the block-based checkout by default; Chapter 4's own cart/checkout material applies to both approaches, since the underlying order and cart logic is identical either way — only the front-end rendering mechanism differs.

## Hands-On Exercises

**EXERCISE 1**

A developer copies WooCommerce's own content-product.php into their theme's woocommerce/ folder to change how each product card looks. Six months later, a WooCommerce update fixes a bug in that same file — but the store's product cards still show the old bug. Explain why.

View solution

**EXERCISE 2**

Explain what a page builder's own WooCommerce widgets are actually doing "under the hood," according to this chapter, and why they don't eliminate the same update-safety consideration that applies to a manually copied template file.

View solution

**EXERCISE 3**

A store owner wants to display a grid of 4 products from a specific category on an ordinary WordPress page, without using the Block Editor. Which mechanism from this chapter should they use, and what would the equivalent Block Editor approach be called?

View solution

### CHAPTER 3 QUICK REFERENCE

- WooCommerce renders every store page from its own template files under `woocommerce/templates/` — a theme overrides one by copying it to an identical path inside `your-theme/woocommerce/`
- A copied template stops receiving WooCommerce's own future updates to that file — Chapter 8's hooks are the lower-risk alternative for smaller changes
- **Storefront** — the official, WooCommerce-native theme, maintained by the same team as WooCommerce itself
- Page builders (Elementor, etc.) provide visual WooCommerce widgets, but still render the same underlying template system — not a separate mechanism
- **Shortcodes** (`[products]`, `[woocommerce_checkout]`, etc.) are the legacy mechanism; **WooCommerce Blocks** are the current, actively-developed one, including a block-based checkout since WooCommerce 8.3
- **Next chapter:** Cart, Checkout & Order Flow

CHAPTER 4 OF 10

# Cart, Checkout & Order Flow

## WordPress E-Commerce with WooCommerce

Chapter 4 · Cart, Checkout & Order Flow

Chapter 3 covered the pages a shopper sees — Cart and Checkout, whether rendered by shortcode or block. This chapter covers what actually happens behind those pages: how a cart is tracked before any order exists, what checkout genuinely does at the moment it's submitted, and the full status lifecycle an order moves through afterward.

## The Cart Is a Session, Not Yet an Order

Adding a product to the cart does not create an order, or any row in `wp_posts` at all. WooCommerce tracks cart contents as **session data** — tied to the shopper's own browser session, not to any permanent database record representing a purchase.

**WHERE THAT SESSION DATA ACTUALLY LIVES**

For a logged-in customer, or a guest with cookies enabled, WooCommerce persists session data (cart contents, applied coupons) in its own dedicated `wp_woocommerce_sessions` table — a custom table, not `wp_posts` or `wp_postmeta`, and not PHP's own default file-based session storage. This is a second real example of WooCommerce reaching for a purpose-built table when the general posts/postmeta pattern genuinely doesn't fit, alongside Chapter 1's own High-Performance Order Storage material.

A practical consequence: an abandoned cart — items added, checkout never completed — leaves no order anywhere in the store's order list. It exists only as a session row, and WooCommerce automatically expires and cleans up old, inactive sessions after a configurable period.

## Checkout: Where a Cart Actually Becomes an Order

Submitting the checkout form is the one moment a session's cart contents become a real, permanent order — a row WooCommerce can report on, email, and track, independent of whether that specific browser session still exists.

1. WooCommerce re-validates the cart — checking each item's current stock and price against what's actually configured right now, not what was cached when the item was added
2. Shipping and tax are calculated (Chapter 6's own territory) based on the shipping address entered

3. A new order is created — a `shop_order` post, or an HPOS order row per Chapter 1's own note on order storage — with an initial status of `pending payment`
4. The chosen payment gateway (Chapter 5) takes over to actually collect payment
5. Once payment is confirmed, the order's status changes automatically, and stock is reduced (see below)

**RE-VALIDATION IS WHY PRICES AND STOCK IN THE CART AREN'T FINAL**

A product's price or stock level can change between "added to cart" and "checkout submitted" — WooCommerce always uses the live, current value at the moment of checkout, not whatever value was shown when the item was first added. A store that changes a price mid-day will see the new price applied to any checkout completed after the change, even for items added to the cart before it.

## Order Statuses — The Full Lifecycle

| STATUS | WHAT IT MEANS |
|---|---|
| Pending payment | Order created, awaiting payment — the default status the moment checkout is submitted |
| On hold | Awaiting payment confirmation from a manual or delayed method (e.g. bank transfer), stock is typically reduced |
| Processing | Payment received; stock reduced; awaiting fulfillment (shipping a physical item, or delivering a service) |
| Completed | Fulfillment finished — order processed and (if applicable) shipped/delivered |
| Cancelled | Cancelled by an admin or the customer — stock is typically restored |
| Refunded | Fully or partially refunded after payment |
| Failed | Payment failed or was declined — no successful payment was ever received |
| Draft / Checkout draft | An order record created automatically as the customer fills in the checkout form, before it's actually submitted — used internally, not shown to store admins as a real order |

**A DIGITAL-ONLY STORE MAY SKIP "PROCESSING" ENTIRELY**

A store selling only downloadable products (with nothing to physically ship) commonly moves an order straight from `Pending payment` to `Completed` once payment succeeds, since there's no separate fulfillment step to track — `Processing` exists specifically to represent the gap between "paid" and "fulfilled," which only exists at all when fulfillment isn't instantaneous.

## Stock Reduction Timing

Chapter 2 covered *what* stock management tracks; this is *when* WooCommerce actually decrements it, a setting under **WooCommerce → Settings → Products → Inventory**:

| OPTION | BEHAVIOR |
|---|---|
| Reduce stock when an order is placed (default) | Stock is decremented as soon as the order reaches `Pending payment`, before payment is confirmed |
| Hold stock (minutes) | An unpaid `Pending payment` order reserves stock only temporarily; if payment isn't completed within this window, the reservation is released and stock returns to being available to other shoppers |

**A REAL, HONEST TENSION WORTH NAMING**

Reducing stock immediately at "Pending payment" protects against overselling a genuinely limited item during a payment delay, but it also means an abandoned, never-paid checkout can temporarily tie up stock another shopper would otherwise have been able to buy — exactly what the "Hold stock" timeout exists to bound. There's no setting that eliminates this trade-off entirely; only how long it's allowed to last.

## Hands-On Exercises

**EXERCISE 1**

A customer adds an item to their cart, then closes the browser tab without checking out. Explain what record of this, if any, exists in the store's own order list, and where the cart's contents were actually being tracked before the tab was closed.

View solution

**EXERCISE 2**

A store owner raises a product's price at 2pm. A customer who added that item to their cart at 1pm finally checks out at 3pm. Which price do they pay, and why, according to this chapter's own checkout re-validation material?

View solution

**EXERCISE 3**

Explain why a store selling only downloadable products might never see an order reach the "Processing" status, connecting your answer to what this chapter says that status specifically represents.

View solution

### CHAPTER 4 QUICK REFERENCE

- A **cart** is session data — for logged-in/cookie-enabled shoppers, stored in the dedicated `wp_woocommerce_sessions` table, not `wp_posts`
- **Checkout** is the moment a session's cart becomes a real order — stock and price are re-validated against current live values, not cached cart values
- **Order statuses:** Pending payment → (On hold /) Processing → Completed, with Cancelled/Refunded/Failed as exit paths, plus an internal Draft/Checkout draft status before submission
- **Stock reduction timing** is configurable — at order placement (default, protects against overselling) versus a "Hold stock" timeout (protects against ties-up-stock abandoned checkouts) — a genuine trade-off, not a solved problem
- **Next chapter:** Payment Gateways

CHAPTER 5 OF 10

# Payment Gateways

## WordPress E-Commerce with WooCommerce

Chapter 5 · Payment Gateways

Chapter 4 named a payment gateway as the piece that "takes over to actually collect payment" once an order is created, without saying how. This chapter covers that mechanism in real detail — and, more importantly, the single distinction that determines whether a store takes on a light or a genuinely heavy compliance burden.

### What a Payment Gateway Plugin Actually Does

A payment gateway is its own separate plugin — WooCommerce Stripe Gateway, WooCommerce PayPal Payments, and similar — built against WooCommerce's own Payment Gateways API. Once activated and configured with API credentials, it adds itself as a selectable payment method at checkout and takes responsibility for one specific job: actually collecting payment for the order Chapter 4 already described being created.

### Hosted & Tokenized Processing — The Distinction That Matters Most in This Chapter

Every reputable modern gateway is built around the same principle: **the store's own server never sees the customer's raw card number.** Two common ways this is achieved:

- **Fully hosted / redirect** — the customer is sent to PayPal's own site (or a fully-hosted Stripe Checkout page) to enter payment details, then returned to the store once payment completes. Card data never touches any page the store itself serves.
- **Embedded, tokenized fields (Stripe Elements)** — card input fields appear directly on the store's own checkout page, but they're rendered inside an isolated frame served by Stripe itself, not the store's own code. Stripe's own JavaScript (`Stripe.js`) converts the entered card details into a one-time **token** before anything is sent to the store's server at all.

```php
<?php
// conceptual server-side flow — the store's server only ever
// receives a token, never a raw card number
$payment_method_id = $_POST['stripe_payment_method_id']; // e.g. "pm_1N..."

$intent = $stripe->paymentIntents->create( array(
    'amount'         => $order->get_total() * 100,
    'currency'       => 'usd',
    'payment_method' => $payment_method_id,
    'confirm'        => true,
) );
?>
```

Notice what's missing entirely: at no point does this code — or anything else running on the store's own server — ever read, log, or store an actual card number. The token stands in for it.

## Why This Distinction Sets the Store's Own PCI-DSS Compliance Burden

**PCI-DSS** (the Payment Card Industry Data Security Standard) applies to any system that stores, processes, or transmits cardholder data. Compliance is self-assessed against one of several **Self-Assessment Questionnaires (SAQs)**, and which one applies depends directly on how much a merchant's own systems are actually exposed to raw card data.

**WHY THE HOSTED/TOKENIZED APPROACH KEEPS COMPLIANCE GENUINELY LIGHT**

A store that fully outsources card handling this way — never having raw cardholder data touch its own server or database at any point — qualifies for one of the lightest available SAQ tiers (commonly SAQ A, or a closely related variant depending on exactly how the fields are embedded). This is the direct, practical payoff of the architecture described above: the store's own compliance burden is small specifically *because* it never becomes a system that stores, processes, or transmits cardholder data in the first place.

**THE MISTAKE THIS CHAPTER EXISTS TO HEAD OFF**

A developer who builds a fully custom checkout form — plain HTML fields for card number, expiry, and CVV, posted directly to the store's own server before ever being sent to a processor — has just built a system that genuinely processes and transmits raw cardholder data. That store no longer qualifies for a light SAQ tier at all; it falls under a far more demanding assessment (such as SAQ D), with real infrastructure-level security requirements most WordPress hosting was never designed to satisfy. This isn't a purely theoretical risk — it's the single most common way a well-meaning custom checkout accidentally creates a serious compliance liability, and it's exactly the mistake Chapter 7's own security chapter assumes has already been avoided.

## Test Mode — Never Skip It

Every major gateway provides a **sandbox** environment with separate test API keys and dedicated test card numbers (Stripe's own `4242 4242 4242 4242` is the best-known example) that simulate a full, realistic checkout with no real money ever moving. WooCommerce's own gateway plugins expose a **Test mode** toggle that swaps live API keys for sandbox ones.

**TWO MIRROR-IMAGE REAL MISTAKES, BOTH WORTH CHECKING FOR BEFORE LAUNCH**

Leaving **Test mode** enabled after launch means real customers' checkout attempts silently fail to charge anything — the store looks broken to a paying customer. Disabling it too early, during active development, means genuine live card numbers can be submitted against a site still being actively debugged. Confirming which mode is active, deliberately, before every deploy is a small habit worth building early.

## Multiple Gateways, One Checkout

A store isn't limited to one gateway — Stripe and PayPal can both be enabled simultaneously, letting the customer choose at checkout. WooCommerce records which gateway processed a given order, along with that gateway's own transaction ID, as order meta — the same underlying mechanism Chapter 2 already covered for storing a product's price or SKU, applied here to an order instead of a product.

## Hands-On Exercises

**EXERCISE 1**

A junior developer proposes building a custom checkout page with plain HTML fields for card number and CVV, posted directly to the store's own PHP backend, arguing it will look more "on-brand" than Stripe's embedded fields. Explain the real, concrete consequence of this choice using this chapter's own material.

View solution

**EXERCISE 2**

Explain what a "token" actually represents in the Stripe payment flow this chapter describes, and why the store's server receiving only a token — rather than a raw card number — is the specific detail that keeps the store's own compliance burden light.

View solution

**EXERCISE 3**

A store launches with Test mode still enabled on its Stripe gateway. Describe what a real customer would actually experience trying to check out, based on this chapter's own explanation of what Test mode does.

📄 View solution

### CHAPTER 5 QUICK REFERENCE

- A payment gateway is its own plugin, built against WooCommerce's Payment Gateways API, responsible for actually collecting the payment Chapter 4's checkout flow creates an order for
- **Hosted/redirect** and **tokenized embedded fields** (Stripe Elements) both keep raw card data off the store's own server entirely
- This is why hosted/tokenized processing qualifies a store for a light PCI-DSS self-assessment tier (commonly **SAQ A** or a close variant) — a custom form posting raw card data to the store's own server does not, and lands in a far heavier tier instead
- **Test mode** swaps live API keys for a gateway's sandbox environment — verify deliberately which mode is active before every launch and every development session
- Multiple gateways can be enabled at once; WooCommerce records which one processed each order, and its transaction ID, as order meta
- **Next chapter:** Shipping & Tax Configuration

CHAPTER 6 OF 10

# Shipping & Tax Configuration

## WordPress E-Commerce with WooCommerce

Chapter 6 · Shipping & Tax Configuration

Chapter 4 named shipping and tax as line items calculated during checkout, deferred to this chapter. Both are configured through the same underlying idea — rules matched against a customer's own address — but the real-world complexity behind each is genuinely different, and worth being honest about rather than glossed over.

## Shipping Zones — Geography-Based Rate Rules

Under **WooCommerce → Settings → Shipping**, a store defines one or more **zones** — named regions (e.g. "Domestic," "Europe," "Rest of World") matched against a customer's country, state, or postcode. Each zone has its own independent, ordered list of shipping methods: Flat rate, Free shipping, Local pickup, or a real-time carrier-calculated rate via an extension.

**ZONE ORDER MATTERS — THE FIRST MATCH WINS**

Zones are evaluated top to bottom, and a customer's address is matched against the *first* zone that fits, not the most specific one. A broad "Europe" zone listed above a narrower "Germany" zone will match every German customer first, and the more specific German-only rates will never actually apply. Order the most specific zones first, the same "order matters" discipline this site's own routing-focused courses have already emphasized in other contexts.

## Shipping Classes — Per-Product Rate Variation Within a Zone

A single zone's Flat Rate method doesn't have to charge every product the same amount. **Shipping classes** — assigned per product, on the Shipping tab of the product editor Chapter 2 already covered — let a Flat Rate method define a different cost for, say, a "Heavy" class versus the default "Standard" class, within the very same zone.

| SHIPPING ZONES ANSWER | SHIPPING CLASSES ANSWER |
| --- | --- |
| "Where is this customer, and which method list applies?" | "Within that method, does this particular product cost more or less to ship?" |

## Tax Classes — Standard, Reduced Rate, Zero Rate

WooCommerce ships with three built-in **tax classes** — `Standard`, `Reduced rate`, and `Zero rate` — assignable per product on the same Shipping/General product tabs. Each class maps to its own rate table under **WooCommerce → Settings → Tax**, with rates entered per country, state, or postcode, mirroring the zone-based matching shipping already uses.

> **WHY MORE THAN ONE TAX CLASS EXISTS AT ALL**
>
> Many real tax jurisdictions don't tax every kind of product at the same rate — groceries or children's clothing are commonly taxed at a reduced or zero rate in jurisdictions that otherwise apply a standard rate to most goods. WooCommerce's three built-in classes exist specifically to represent that kind of real, common distinction, and a store can define further custom classes if its own jurisdiction needs more than three.

## Real-World Complexity: Nexus

Deciding *where* a US-based store must actually collect sales tax at all is not simply "wherever the store is located." **Nexus** is the legal connection between a business and a taxing jurisdiction that creates an obligation to collect tax there.

> **A REAL, SPECIFIC LEGAL FACT WORTH CITING PRECISELY**
>
> Nexus can be established physically (an office, warehouse, or employee in a state) or economically. Since the U.S. Supreme Court's 2018 decision in *South Dakota v. Wayfair, Inc.* — which overturned the prior physical-presence-only standard from *Quill Corp. v. North Dakota* (1992) — a state can also require tax collection once a remote seller crosses a defined threshold of sales or transactions in that state, even with no physical presence there at all. Most US states have since adopted their own version of this economic-nexus threshold, and the specific thresholds vary state by state.

## Real-World Complexity: VAT

Selling into the European Union introduces a different model. **VAT** (Value Added Tax) is generally charged at the *buyer's* own country rate, not the seller's — meaning a single EU-facing store may owe tax at dozens of different rates depending on where each individual customer is located. Since July 2021, the EU's **One Stop Shop (OSS)** scheme (building on an earlier 2015 scheme limited to digital services) lets a seller report and remit VAT for sales across the entire EU through a single registration, rather than registering separately in every member state.

> **THIS CHAPTER TEACHES WOOCOMMERCE'S OWN CONFIGURATION MECHANICS — NOT TAX LAW**
>
> Nexus thresholds, VAT registration requirements, and which products qualify for a reduced or zero rate are genuine legal questions, they change over time, and getting them wrong carries real financial and legal consequences. This

chapter explains how WooCommerce's own tax-class and rate-table system works mechanically; it is not a substitute for a real accountant or tax professional confirming what a specific store actually owes, where.

## Automated Tax Calculation

Given that complexity, most real stores don't maintain their own rate tables by hand. **WooCommerce Tax** (a free extension from the same team behind WooCommerce) and third-party services like Avalara or TaxJar automatically calculate the correct rate for each order based on the customer's address and current rules, and update as rates or thresholds change — removing the need to manually track a rate table across every jurisdiction a store might sell into.

## Hands-On Exercises

**EXERCISE 1**

A store defines a "Europe" shipping zone listed above a more specific "Germany" zone with cheaper domestic-German rates. Explain what a German customer will actually be charged for shipping, and why, using this chapter's own zone-matching rule.

View solution

**EXERCISE 2**

Explain the difference between what a shipping zone controls and what a shipping class controls, using this chapter's own two-question framing.

View solution

**EXERCISE 3**

Explain what changed about US sales tax obligations after South Dakota v. Wayfair, Inc., according to this chapter, and why a store with no physical presence in a state could still owe tax there as a result.

View solution

### CHAPTER 6 QUICK REFERENCE

- **Shipping zones** match a customer's address to an ordered list of methods — the first matching zone wins, so order the most specific zones first

- **Shipping classes** let one method charge different products differently within the same zone
- **Tax classes** (Standard/Reduced rate/Zero rate) map to per-jurisdiction rate tables, mirroring shipping's own zone-based matching
- **Nexus** — the legal trigger for a US tax-collection obligation — can be physical or, since *South Dakota v. Wayfair* (2018), purely economic based on sales volume in a state
- **VAT** is generally charged at the buyer's own EU country rate; the OSS scheme (since July 2021) allows single-registration reporting across the EU
- This chapter is not tax advice — real nexus/VAT obligations require a qualified accountant or tax professional
- **Next chapter:** Security for E-Commerce Sites

CHAPTER 7 OF 10

# Security for E-Commerce Sites

## WordPress E-Commerce with WooCommerce

Chapter 7 · Security for E-Commerce Sites

*WordPress Intermediate/Advanced 8* taught four mechanisms — capability checks, nonces, output escaping, and `$wpdb->prepare()` — as WordPress's own concrete implementations of general security principles this site already covered in depth. Nothing in this chapter is a new vulnerability class. What's new is the stakes: applied to a checkout page or an account form, the same four mechanisms stand between an attacker and real money, real payment metadata, and real customer PII — not a defaced blog post.

### HTTPS: Non-Negotiable for a Store

*HTTPS/TLS Fundamentals* already covered why encryption in transit matters generally. For a store, it stops being a best practice and becomes a hard requirement: Chapter 5's own tokenized payment fields (Stripe Elements, PayPal) require a secure context to function at all — modern browsers block or restrict the APIs these gateways depend on over plain HTTP, and PCI-DSS itself requires TLS for any transmission of payment-related data. A store running checkout over HTTP isn't taking a small risk; in practice, its payment gateway simply won't work correctly.

### XSS on Reviews & Account Fields

*XSS — Cross Site Scripting*'s general lesson — encode output for its specific context, never trust rendered user input — applies directly to two store-specific surfaces: product reviews (user-submitted text, rendered back to every visitor of that product page) and account fields like a saved shipping address (rendered back to the customer, and often to store staff processing the order).

```php
<?php
// vulnerable: a customer's own saved address, echoed with no escaping
echo '<p>' . $customer_address . '</p>';

// fixed: WordPress Intermediate/Advanced 8's own esc_html(), applied here
echo '<p>' . esc_html( $customer_address ) . '</p>';
?>
```

**A STORE-SPECIFIC REASON THIS MATTERS MORE THAN A TYPICAL COMMENT FIELD**

An unescaped review or account field being rendered on an order-processing screen means store staff — often with far more system access than an ordinary site visitor — are the ones whose browser executes any injected script, potentially against an admin session with real order-management and refund capabilities.

## CSRF on Checkout & Account Actions — Nonces Again

*CSRF — Cross-Site Request Forgery*'s general defense — a token proving a request genuinely originated from the expected page — is exactly what protects "Place order" and account-changing actions. WooCommerce's own checkout form includes a nonce field (commonly named `woocommerce-process-checkout-nonce`), verified server-side before an order is ever created — the same `wp_nonce_field()` / `wp_verify_nonce()` pair *WordPress Intermediate/Advanced 8* already covered, doing real work at the single most consequential form on the entire site.

**CUSTOM CHECKOUT FIELDS INHERIT THIS PROTECTION AUTOMATICALLY**

Any custom field added to the checkout form (Chapter 8 covers adding these via hooks) submits inside the same form, protected by the same nonce, as long as it isn't processed through a separate, custom AJAX endpoint that skips WooCommerce's own verification — a real, common mistake worth watching for specifically when extending checkout.

## SQL Injection in Custom Order Queries

*SQL Injections and Defences*'s general defense — never concatenate user input into a SQL string — applies directly the moment a store adds any custom feature that queries orders by user-supplied criteria, such as an "order lookup by email" tool.

```php
<?php
// vulnerable: customer-supplied email concatenated directly into SQL
$results = $wpdb->get_results(
    "SELECT * FROM {$wpdb->prefix}wc_orders WHERE billing_email = '" . $_GET['email'] . "'"
);

// fixed: WordPress Intermediate/Advanced 8's own $wpdb->prepare(), applied here
$results = $wpdb->get_results(
    $wpdb->prepare(
        "SELECT * FROM {$wpdb->prefix}wc_orders WHERE billing_email = %s",
        $_GET['email']
    )
);
?>
```

**THIS APPLIES WHETHER OR NOT THE STORE USES HPOS**

Chapter 1 named High-Performance Order Storage as moving order data into dedicated tables like `wc_orders`. Custom code querying either the older `wp_posts`-based order storage or the newer HPOS tables needs `$wpdb->prepare()` exactly the same way — the underlying storage location changed, but the injection risk from unsanitized concatenation didn't.

## Broken Access Control: Viewing Another Customer's Order

A customer's own **My Account → Orders** page correctly shows only their own orders, because WooCommerce's own code checks that the logged-in user actually owns each order before displaying it. Custom code that adds an order-lookup feature — say, a URL like `/order-status/?order_id=482` — can easily reintroduce exactly the gap *WordPress Intermediate/Advanced 8*'s own capability-check material warned about, if it fetches order #482 and displays it without confirming the currently logged-in user actually owns order #482.

**HIDING THE LINK IS NOT THE SAME FIX HERE EITHER**

Exactly as *WordPress Intermediate/Advanced 8* warned about a hidden "Delete Post" button, not linking directly to other customers' order IDs doesn't stop someone from simply guessing or incrementing the number in the URL. The order-fetching code itself must check ownership — comparing the order's own customer ID against the currently logged-in user's ID — independent of what links are or aren't shown anywhere in the interface.

## A Compromised Checkout Page Is a Fraud Vector, Not Just a Defacement

**A REAL, IMPORTANT LIMIT ON CHAPTER 5'S OWN TOKENIZATION MATERIAL**

Chapter 5 explained that Stripe's tokenized fields keep raw card numbers off the store's own server. That protects the *server* — it does not automatically protect the *browser*. Malicious JavaScript injected into a checkout page — through an unpatched, vulnerable plugin, exactly the risk *WordPress Fundamentals 6* and *WordPress Fundamentals 10* already warned about — can still read keystrokes or overlay a fake payment form directly in the customer's own browser, before Stripe.js ever gets a chance to tokenize anything. This general pattern (real-world attacks against e-commerce checkout pages of exactly this shape are commonly called "Magecart-style" attacks, after the group that popularized it) is precisely why keeping every plugin patched and vetted matters even more on a store than on an ordinary blog — Chapter 5's own architecture doesn't make plugin security optional.

## The Synthesis: Same Vulnerabilities, Store-Specific Stakes

**TYING IT BACK TO OWASP TOP 10 DIRECTLY**

Every issue this chapter covers maps directly onto categories *OWASP Top 10* already taught in general — Broken Access Control (the order-lookup example), Injection (the SQL example), Security Misconfiguration and Vulnerable/Outdated Components (the plugin-patching material, echoing *WordPress Fundamentals 10*'s own update-

risk chapter), and Cryptographic Failures (the HTTPS material). None of it is new to e-commerce specifically. What changes on a store is what a successful exploit is actually worth to an attacker: real payment fraud, a real customer PII breach with genuine legal exposure under regimes like GDPR, and direct revenue loss — not merely reputational embarrassment.

## Hands-On Exercises

**EXERCISE 1**

A developer builds an order-status lookup page at /order-status/?order_id=N, and fetches/displays whatever order matches that ID with no further check. Explain the exploit this leaves open, and the specific fix, referencing this chapter's own comparison to WordPress Intermediate/Advanced 8.

View solution

**EXERCISE 2**

A store owner argues that since they use Stripe's tokenized checkout fields, a vulnerable plugin on their site can't put customer card data at risk. Explain why this chapter says that reasoning is incomplete.

View solution

**EXERCISE 3**

Explain why this chapter says none of the vulnerability classes it covers are new to e-commerce specifically, while also arguing this chapter is still necessary and not just a repeat of WordPress Intermediate/Advanced 8.

View solution

### CHAPTER 7 QUICK REFERENCE

- **HTTPS** is a hard requirement for a store — Chapter 5's own payment gateways need a secure context to function at all
- **XSS** — escape reviews and saved account fields on output, especially since store-staff sessions often carry real order-management privileges
- **CSRF** — WooCommerce's own checkout form already uses a nonce; custom AJAX-based checkout additions must not bypass it
- **SQL injection** — any custom order query, against either legacy post-based storage or HPOS, needs `$wpdb->prepare()`
- **Broken access control** — any custom order-lookup feature must verify the requesting user actually owns that order, not just avoid linking to it

- **Tokenization protects the server, not the browser** — a compromised plugin can still enable client-side, "Magecart-style" card skimming even with Stripe/PayPal tokenization in place
- None of these are new vulnerability classes — what changes on a store is the real-world value of a successful exploit: fraud, PII breach, and direct revenue loss
- **Next chapter:** Extending WooCommerce: Hooks & Custom Functionality

CHAPTER 8 OF 10

# Extending WooCommerce: Hooks & Custom Functionality

## WordPress E-Commerce with WooCommerce

Chapter 8 · Extending WooCommerce: Hooks & Custom Functionality

Chapter 3 warned that copying a template file forks it, permanently losing that file's future WooCommerce updates — and pointed forward to hooks as the lower-risk alternative. This chapter is that payoff: *WordPress Intermediate/Advanced 5*'s own actions-and-filters material, applied to WooCommerce's own large, separate set of hooks.

### Actions vs. Filters, Recapped for WooCommerce

| ACTIONS | FILTERS |
| --- | --- |
| "Do something at this point" — no return value expected | "Modify this value and return it" — the hook is useless if nothing is returned |
| `add_action( 'hook_name', 'callback' )` | `add_filter( 'hook_name', 'callback' )` |
| e.g. sending a notification once an order reaches a given status | e.g. adjusting a cart item's price before totals are calculated |

This is exactly the mechanism *WordPress Intermediate/Advanced 5* already taught — nothing new conceptually. What's new is the specific set of hooks: WooCommerce fires hundreds of its own action and filter hooks throughout the cart, checkout, product display, and order lifecycle, entirely separate from WordPress core's own hooks.

### Finding the Right Hook

WooCommerce publishes an official hook reference documenting its own actions and filters by name and by the data each one provides. In practice, confirming a hook actually fires when and how expected is often done directly and quickly — temporarily adding a logging line inside a candidate callback while testing is a normal, standard part of the workflow, not a sign of doing it wrong.

### A Real Example: A Bulk-Discount Pricing Filter

A common, genuinely useful customization: automatically discounting a cart item once its quantity reaches a threshold. The standard, reliable pattern for adjusting cart-item prices hooks into `woocommerce_before_calculate_totals` — an action, despite modifying a price, because it acts on the whole cart object rather than returning a single filtered value.

```php
<?php
add_action( 'woocommerce_before_calculate_totals', 'apply_bulk_discount', 20, 1 );

function apply_bulk_discount( $cart ) {
    if ( is_admin() && ! defined( 'DOING_AJAX' ) ) {
        return; // don't run this on ordinary wp-admin page loads
    }

    foreach ( $cart->get_cart() as $cart_item ) {
        if ( $cart_item['quantity'] >= 5 ) {
            $original_price = $cart_item['data']->get_price();
            $cart_item['data']->set_price( $original_price * 0.9 ); // 10% off
        }
    }
}
?>
```

`$cart_item['data']` is the product object for that line item — the same product object *WordPress Intermediate/Advanced 4* already covered as a custom post type instance, reached here through the cart rather than a direct query. Calling `set_price()` on it changes the price used for totals, tax, and the order eventually created at checkout, without ever touching the product's own stored price in `wp_postmeta`.

**THE ADMIN/AJAX GUARD AT THE TOP ISN'T OPTIONAL**

`woocommerce_before_calculate_totals` also fires on ordinary wp-admin page loads — without the guard shown above, this same code could run in contexts with no real cart to discount, or interfere with the admin order-editing screen. This is a small, real, commonly-forgotten detail specific to this exact hook.

## A Real Example: An Action for a Side Effect

Not every customization changes a value — some just need to *do* something at a specific point. Notifying an internal fulfillment system once an order is marked complete is a genuine action, not a filter:

```php
<?php
add_action( 'woocommerce_order_status_completed', 'notify_fulfillment_team' );

function notify_fulfillment_team( $order_id ) {
    $order = wc_get_order( $order_id );

    wp_mail(
        'fulfillment@example.com',
        sprintf( 'Order #%d ready for fulfillment', $order_id ),
        $order->get_formatted_billing_full_name()
    );
}
?>
```

`wc_get_order()` returns a full order object — the same underlying order data Chapter 4 described being created at checkout, now available inside a hook triggered by that same order's own status change.

## Where to Put This Code

*WordPress Intermediate/Advanced 5* already made the case for a small custom plugin over a theme's own `functions.php` — a plugin survives a future theme change, `functions.php` doesn't. That reasoning applies here without modification.

**A WOOCOMMERCE-SPECIFIC TIMING GOTCHA, WORTH NAMING PRECISELY**

Code that references WooCommerce's own classes or functions must not run before WooCommerce itself has finished loading — a fatal error results if a class like `WC_Cart` is referenced before it exists yet. Wrapping WooCommerce-dependent code inside a `plugins_loaded` hook, or checking `class_exists( 'WooCommerce' )` first, avoids this specific failure mode, which otherwise tends to appear intermittently depending on plugin load order.

## Hands-On Exercises

**EXERCISE 1**

Explain why the bulk-discount example hooks into an action (woocommerce_before_calculate_totals) rather than a filter, even though its actual job is changing a price.

View solution

**EXERCISE 2**

A developer removes the is_admin()/DOING_AJAX guard from the top of the bulk-discount function, believing it's unnecessary boilerplate. Explain what problem this chapter warns this could cause.

View solution

**EXERCISE 3**

A developer's custom plugin throws a fatal error referencing an undefined WC_Cart class. Explain the likely cause, and the two fixes this chapter names for it.

View solution

### CHAPTER 8 QUICK REFERENCE

- Same actions/filters mechanism as *WordPress Intermediate/Advanced 5* — WooCommerce just fires its own large, separate set of hooks
- **woocommerce_before_calculate_totals** + `set_price()` — the standard pattern for custom cart-item pricing (an action, since it acts on the whole cart object)
- **woocommerce_order_status_completed** — a real example of an action used for a side effect (a notification), not a value change
- Put custom hooks in a small site-specific plugin, not `functions.php` — the same guidance *WordPress Intermediate/Advanced 5* already gave
- Guard any WooCommerce-dependent code with `class_exists( 'WooCommerce' )` or a `plugins_loaded` hook to avoid a fatal error from code running before WooCommerce itself has loaded
- **Next chapter:** Performance at Scale

CHAPTER 9 OF 10

# Performance at Scale

## WordPress E-Commerce with WooCommerce

Chapter 9 · Performance at Scale

*WordPress Intermediate/Advanced 9* already covered caching and mapped it to *Performance & Core Web Vitals*'s own LCP/INP/CLS metrics — for an ordinary page, that material applies to a WooCommerce store unchanged. This chapter covers the one genuine complication e-commerce adds: some of a store's own pages can't simply be cached the way a blog post can, and WooCommerce has a specific, real mechanism for handling that.

### Why Full-Page Caching Can't Simply Be Applied Everywhere

Full-page caching, as *WordPress Intermediate/Advanced 9* covered it, works by serving one pre-generated copy of a page's HTML to every visitor — genuinely safe and effective for a blog post, whose content is identical no matter who requests it. Cart, Checkout, and My Account pages break that assumption entirely: their content is different for every single visitor, by design.

> **WHAT NAIVELY CACHING THESE PAGES WOULD ACTUALLY DO**
>
> Serving a full-page cached copy of the Cart page would show one visitor a different visitor's own cart contents — not a performance bug, but a genuine data-leak bug. The same problem applies to Checkout (showing stale totals or another customer's billing details) and My Account (showing another customer's order history). These pages must always be generated fresh, per visitor, every time.

### WooCommerce's Own Solution: Cart Fragments

The naive fix — excluding Cart, Checkout, and My Account entirely from caching — solves the data-leak problem but still leaves one gap: the small "mini-cart" widget (an item-count icon in the header, for example) commonly appears on *every* page, including the Shop and product pages that are otherwise perfectly safe to cache. If those pages are cached, that widget's cached HTML would show whatever cart count happened to exist when the page was cached — wrong for almost every subsequent visitor.

WooCommerce solves this with **Cart Fragments**: after a cached page finishes loading, a small AJAX request (`wc-cart-fragments.js`, built on the `woocommerce_add_to_cart_fragments` filter) fetches just the visitor-specific pieces — the mini-cart's own current markup — and swaps them into the already-loaded page. The

surrounding page stays fully cacheable; only the small fragment that genuinely needs to be personal is fetched fresh.

> **TWO LAYERS WORKING TOGETHER, NOT ONE SOLVING EVERYTHING**
> A production caching setup typically combines both techniques: a caching plugin configured to exclude Cart/Checkout/My Account from the cache entirely (since those pages are inherently personal throughout), plus Cart Fragments handling the smaller personalization gap on pages that otherwise remain fully cached. Neither technique alone is the complete picture.

## Object Caching at Catalog Scale

*WordPress Intermediate/Advanced 9*'s own object-caching material (Redis/Memcached, avoiding repeated identical database queries within and across requests) matters more for a store than for an ordinary blog, simply because of how much more repeated data lookup a store's own pages generate — product data, stock status, and pricing are read constantly across the Shop page, product pages, and the cart itself, often for the very same products, request after request.

> **WHERE HPOS'S OWN PERFORMANCE MOTIVATION BECOMES CONCRETE**
> Chapter 1 named High-Performance Order Storage as purpose-built tables that scale better than the general `wp_posts` / `wp_postmeta` pattern for a busy store's own order-query patterns. This chapter is where that motivation stops being abstract: a large catalog with many variable products (Chapter 2) and a meaningful order volume is exactly the scale at which the general posts/postmeta pattern's own query cost becomes genuinely noticeable — precisely the pain HPOS was built to reduce.

## Core Web Vitals on Product Pages Specifically

*Performance & Core Web Vitals*'s three metrics apply to a store with their own specific, common failure points:

| METRIC | COMMON STORE-SPECIFIC CAUSE |
|---|---|
| **LCP** (Largest Contentful Paint) | The main product image is very often the page's own largest element — unoptimized product photography is a frequent, direct cause of a poor LCP score |
| **INP** (Interaction to Next Paint) | A sluggish response from the "Add to Cart" button, especially on AJAX-based add-to-cart flows competing with Cart Fragments' own request for the same connection |
| **CLS** (Cumulative Layout Shift) | A Variable product's price and stock status changing as the shopper selects different attributes (Chapter 2) — reserving stable layout space for these values avoids a visible shift each time a selection changes |

## Hands-On Exercises

**EXERCISE 1**

A developer configures a full-page caching plugin to cache every page on the site with no exclusions, arguing "faster is always better." Explain the specific, concrete bug this would create on the store's Checkout page.

View solution

**EXERCISE 2**

A store excludes Cart, Checkout, and My Account from its caching plugin, but leaves the Shop and product pages cached. A shopper notices the mini-cart icon in the header sometimes briefly shows the wrong item count for a split second after a page loads, then corrects itself. Explain what's happening, using this chapter's own Cart Fragments material.

View solution

**EXERCISE 3**

Explain why a Variable product's page is more prone to a poor CLS score than a Simple product's page, and what this chapter recommends to reduce that risk.

View solution

### CHAPTER 9 QUICK REFERENCE

- Full-page caching can't apply to Cart, Checkout, or My Account — their content is inherently visitor-specific, and caching them would leak one customer's data to another
- **Cart Fragments** (`wc-cart-fragments.js`) solve the remaining gap on otherwise-cacheable pages — an AJAX call refreshes just the personal mini-cart fragment after the cached page loads
- Object caching matters more at catalog scale, since product/stock/price data is repeatedly re-read across Shop, product, and cart pages
- HPOS's own performance motivation (Chapter 1) becomes concrete at exactly this scale — large catalogs and meaningful order volume
- **LCP** ← product image optimization; **INP** ← Add to Cart responsiveness; **CLS** ← reserving layout space for variation-dependent price/stock changes
- **Next chapter:** Capstone — Building and Securing a Complete Online Store

CHAPTER 10 OF 10

# Capstone: Building and Securing a Complete Online Store

## WordPress E-Commerce with WooCommerce

Chapter 10 · Capstone — Building and Securing a Complete Online Store

Nine chapters have built the pieces, one at a time. This capstone assembles them into one real, working store — **Wick & Ember**, a small candle business — from catalog through a hardened, launch-ready checkout, closing this course and the WooCommerce/e-commerce gap *WordPress Intermediate/Advanced 10*'s own honest scope note deliberately left open.

## 1. Installing WooCommerce & Building the Catalog

WooCommerce is installed and activated the standard plugin way (Chapter 1), and the setup wizard's store details are filled in. The catalog is built from both product types Chapter 2 covered:

| PRODUCT | TYPE | DETAIL |
|---|---|---|
| Vanilla Bean, Single Wick | Simple | One price, SKU `WE-VAN-1W`, stock managed with a low-stock threshold of 10 |
| Signature Scent Trio | Variable | Attributes Scent (Vanilla/Cedar/Citrus) × Size (Small/Large) — 6 variations, each with its own SKU/price/stock, per Chapter 2's own parent/`product_variation` child relationship |

## 2. Store Design

The Storefront theme is activated (Chapter 3), and the homepage places a Products block configured to show the store's featured items — the modern equivalent of the `[products]` shortcode Chapter 3 also covered, chosen here since the homepage is being built in the Block Editor.

## 3. Verifying the Cart & Checkout Flow

Before connecting real payment, a full test purchase confirms Chapter 4's own order lifecycle end to end: adding items creates only session data in `wp_woocommerce_sessions`, submitting checkout creates a real order at `Pending payment`, and — since Wick & Ember sells physical, shippable candles rather than downloads — the order is expected to pass through `Processing` before reaching `Completed`, exactly the gap Chapter 4 explained that status exists to represent.

## 4. Configuring Payment: Stripe in Test Mode

The Stripe gateway plugin is configured with **Test mode** enabled (Chapter 5) and a full checkout is run using Stripe's own `4242 4242 4242 4242` test card. Chapter 5's own PCI-DSS material is confirmed directly: Stripe's embedded fields tokenize the card before anything reaches Wick & Ember's own server, keeping the store's compliance burden at the lighter SAQ A tier rather than the heavy SAQ D tier a custom card form would require.

## 5. Shipping & Tax

A single "Domestic" shipping zone is configured with a Flat Rate method, plus a "Heavy" shipping class (Chapter 6) applied to the Signature Scent Trio's larger candles, priced higher than the Standard-class single-wick candle. Tax rates are configured for Wick & Ember's own home state, with a note — per Chapter 6's own honest disclaimer — to confirm actual multi-state nexus obligations with a real accountant before expanding sales significantly beyond that state.

## 6. Hardening a Custom Order-Lookup Feature

Wick & Ember wants a simple order-status lookup page for customers who checked out as guests, without a My Account login. Built naively, this is exactly Chapter 7's own broken-access-control example — fixed here with an explicit ownership check, alongside escaping and a parameterized query:

```php
<?php
function guest_order_lookup() {
    $order_id = absint( $_GET['order_id'] );
    $email    = sanitize_email( $_GET['email'] );

    $order = wc_get_order( $order_id );

    // Chapter 7's own ownership check — the order's billing email must
    // match the email the requester actually supplied, not just any valid ID
    if ( ! $order || $order->get_billing_email() !== $email ) {
        return 'Order not found.';
    }

    // Chapter 7's own escaping — order status came from the database,
    // but it is still output, so it is still escaped
    return esc_html( $order->get_status() );
}
?>
```

Every guest order lookup now requires *both* a valid order ID *and* the matching billing email — an attacker guessing sequential order IDs alone can no longer view another customer's order status.

## 7. A Wholesale Bulk-Discount Hook

A local gift shop wants to buy candles wholesale, 5 or more at a time, at a 10% discount. Chapter 8's own bulk-discount filter is added, unchanged, via a small site-specific plugin rather than the theme's `functions.php` — guarded, per Chapter 8, against firing outside a real cart context:

```php
<?php
add_action( 'woocommerce_before_calculate_totals', 'apply_bulk_discount', 20, 1 );

function apply_bulk_discount( $cart ) {
    if ( is_admin() && ! defined( 'DOING_AJAX' ) ) {
        return;
    }
    foreach ( $cart->get_cart() as $cart_item ) {
        if ( $cart_item['quantity'] >= 5 ) {
            $original_price = $cart_item['data']->get_price();
            $cart_item['data']->set_price( $original_price * 0.9 );
        }
    }
}
?>
```

## 8. Verifying Caching & Core Web Vitals Before Launch

- The caching plugin is configured to exclude Cart, Checkout, and My Account entirely, per Chapter 9's own data-leak warning
- The mini-cart's Cart Fragments are confirmed working — adding an item from a cached Shop page updates the header's item count without a full page reload
- Product photos are confirmed compressed and appropriately sized, addressing Chapter 9's own LCP concern before launch
- The Signature Scent Trio's variation-selector area is confirmed to reserve stable space for price/stock text, avoiding the CLS risk Chapter 9 named for Variable products specifically

> **ONLY NOW — SWITCHING STRIPE OUT OF TEST MODE**
>
> With the catalog built, checkout verified, security hardened, and caching configured correctly, Stripe's Test mode is finally switched off and replaced with live API keys — deliberately the very last step, so nothing still being actively verified could ever accidentally process a real charge.

## Chapter Attribution

| PIECE | SOURCE CHAPTER |
|---|---|
| WooCommerce as a plugin, not a separate platform; installation | Chapter 1 |
| Simple and Variable products, SKUs, stock management | Chapter 2 |
| Storefront theme, the Products block | Chapter 3 |
| The session-based cart, checkout, and order-status lifecycle | Chapter 4 |
| Tokenized Stripe checkout, Test mode, PCI-DSS scope | Chapter 5 |
| Shipping zones, shipping classes, tax classes, nexus disclaimer | Chapter 6 |
| Ownership-checked, escaped, parameterized order lookup | Chapter 7 |
| The bulk-discount hook, in a site-specific plugin | Chapter 8 |
| Cache exclusions, Cart Fragments, LCP/CLS checks | Chapter 9 |

## Honest Scope Note

**WHAT THIS CAPSTONE DELIBERATELY DOESN'T COVER**

- No multi-currency or multi-language support
- No marketplace/multi-vendor extensions (a single-seller store only)
- No subscriptions or memberships extensions
- No headless/REST-API-driven storefront — a traditional, theme-rendered store throughout

Each is a legitimate, genuinely separate future topic — not silently assumed solved here.

## Hands-On Exercises

**EXERCISE 1**

Explain why the guest order-lookup function checks both the order ID and the billing email, rather than the order ID alone, referencing this capstone's own Chapter 7 material.

View solution

**EXERCISE 2**

Explain why this capstone switches Stripe out of Test mode only as the very last step, after every other piece of the store has already been verified.

View solution

**EXERCISE 3**

Explain why the Signature Scent Trio (a Variable product) needed a specific Core Web Vitals check in Step 8 that the Vanilla Bean Single Wick (a Simple product) didn't, referencing this course's own Chapter 9 material.

View solution

### CHAPTER 10 QUICK REFERENCE — COURSE & TRACK COMPLETE

- A real store combining WooCommerce's plugin architecture, a mixed product catalog, store design, verified checkout, tokenized payment, shipping/tax, hardened custom code, a real extensibility hook, and a pre-launch performance check
- The order-lookup fix is the single most security-sensitive piece — ownership check, then escaped output, then a parameterized query if a raw SQL lookup were ever needed instead
- Test mode is switched off only as the final step, deliberately, once everything else is already verified
- This closes the WooCommerce/e-commerce gap *WordPress Intermediate/Advanced 10*'s own honest scope note left open, completing the full WordPress track: *WordPress Fundamentals* (10 ch.), *WordPress Intermediate/Advanced* (10 ch.), and *WordPress E-Commerce with WooCommerce* (10 ch.) — 30 chapters in total