E-commerce Store: Design Document

Overview

A full-stack e-commerce platform that enables customers to browse products, manage shopping carts, and complete purchases while providing merchants with inventory management capabilities. The key architectural challenge is maintaining data consistency across user sessions, inventory updates, and order processing while handling concurrent operations safely.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** Foundation for all milestones - establishes the domain understanding and architectural context

Building an e-commerce platform presents a unique set of challenges that bridge the gap between familiar physical retail operations and complex distributed systems engineering. While the core business concepts of products, shopping carts, and transactions are universally understood, translating these into reliable, concurrent digital systems requires careful consideration of data consistency, user state management, and business rule enforcement.

This section establishes the foundational understanding of e-commerce systems by grounding technical concepts in familiar real-world analogies, identifying the core engineering challenges that differentiate e-commerce from simpler web applications, and evaluating architectural approaches that have evolved to address these challenges.

Physical Store Mental Model

Understanding e-commerce systems becomes significantly easier when we map digital concepts to their physical retail counterparts. This mental model provides intuitive understanding before diving into technical implementation details.

Consider a traditional brick-and-mortar bookstore. Customers enter the store, browse through categorized shelves, examine individual books, place selected items in a shopping basket, and eventually proceed to the cashier for checkout. Throughout this process, the store maintains inventory levels, tracks customer activity, and ensures that popular items don't run out of stock unexpectedly.

The Product Catalog functions like the store's organized shelving system. Books are categorized by genre, author, and subject matter, with clear labeling and pricing. The store layout enables customers to navigate efficiently from broad categories to specific items. Digital catalogs mirror this organization through hierarchical category structures, search functionality, and detailed product pages. Just as physical stores use signage and organization to guide customer discovery, digital catalogs employ filtering, sorting, and recommendation systems to help users find relevant products.

The Shopping Cart represents the customer's temporary selection state during their shopping journey. In physical stores, customers might carry a basket or push a cart, accumulating items as they browse. They can examine their current selections, remove unwanted items, or modify quantities before heading to checkout. Digital shopping carts serve identical purposes but must additionally handle persistence across browser sessions, price updates when products change, and availability validation when items become out of stock.

User Authentication parallels the store's customer relationship management. Regular customers might have membership cards or loyalty accounts that track purchase history and provide personalized experiences. Store employees recognize frequent customers

and can access their preferences or previous purchases. Digital authentication systems serve similar functions, maintaining user identity across sessions, storing personal preferences, and enabling features like order history and saved addresses.

The Checkout Process mirrors the cashier station where customers finalize purchases. Physical checkout involves scanning items, calculating totals with taxes and discounts, collecting payment information, updating inventory records, and providing receipts. Digital checkout must orchestrate these same steps while handling additional complexities like address validation, payment processing integration, and order confirmation delivery.

However, the digital environment introduces complexities that have no direct physical analogies. Multiple customers can simultaneously attempt to purchase the last item in stock, requiring sophisticated concurrency control. Shopping carts must persist across browser crashes and device switches. Price changes must be communicated and handled gracefully. Session management must balance security with user experience across potentially unreliable network connections.

Physical Store Concept	Digital E-commerce Equivalent	Additional Digital Complexity
Organized shelving and product displays	Product catalog with categories and search	Real-time inventory updates, image optimization, search indexing
Shopping basket or cart	Digital shopping cart with session persistence	Cross-device synchronization, price staleness, session expiry
Customer recognition and membership	User authentication and profile management	Password security, session hijacking, account recovery
Cashier checkout process	Automated checkout and order processing	Payment integration, inventory race conditions, transaction atomicity
Inventory management	Real-time stock tracking and availability	Concurrent inventory updates, overselling prevention, backorder handling
Store hours and capacity	24/7 availability and scalability	Load balancing, database scaling, global content delivery

Technical Challenges

E-commerce systems present several technical challenges that distinguish them from simpler web applications. These challenges stem from the need to maintain data consistency, handle concurrent operations, and provide reliable user experiences across potentially unreliable network connections.

Concurrent Inventory Management represents the most critical technical challenge in e-commerce systems. Unlike content management systems where multiple users can read the same article simultaneously without conflict, e-commerce platforms must carefully coordinate access to limited inventory. Consider a scenario where two customers simultaneously attempt to purchase the last item in stock. The system must ensure that only one customer successfully completes the purchase while gracefully handling the other customer's disappointment.

This challenge extends beyond simple race conditions. Inventory levels must remain consistent across multiple database operations: adding items to carts, removing items when carts are abandoned, updating quantities during checkout, and handling returns or cancellations. Each operation potentially affects global inventory state, requiring careful transaction management and conflict resolution strategies.

> The fundamental tension in inventory management is between user experience and data consistency. Optimistic approaches provide better user experience but risk overselling, while pessimistic approaches ensure accuracy but can frustrate customers with frequent "out of stock" messages.

Session State Management introduces complexity absent from stateless web applications. Shopping carts represent temporary user state that must persist across browser sessions, survive network interruptions, and synchronize across multiple devices. Users expect their shopping cart contents to remain available whether they return minutes later or days later, whether they switch from desktop to mobile, or whether they experience network connectivity issues during their shopping session.

This persistence requirement creates several technical challenges. Cart data must be stored reliably, but not every cart results in a purchase, leading to significant storage overhead from abandoned carts. Price information in carts can become stale as product prices change, requiring strategies to handle price updates gracefully. Session expiration policies must balance security concerns with user experience expectations.

Transaction Consistency and Atomic Operations become critical when checkout operations must coordinate multiple system components. A successful order requires atomically updating inventory levels, creating order records, clearing shopping carts, and potentially triggering external payment processing and fulfillment systems. Partial failures during this process can leave the system in inconsistent states where customers are charged but inventory isn't decremented, or orders are created but payment processing fails.

These atomic operations must handle not only database-level transaction management but also integration with external systems that may have different consistency guarantees and failure modes. Payment processors, inventory management systems, and shipping providers each introduce potential points of failure that the e-commerce platform must handle gracefully.

Price and Currency Precision requires careful consideration of numeric representation and calculation accuracy. E-commerce systems must handle monetary calculations with perfect precision to avoid rounding errors that could accumulate over many transactions. Tax calculations, discount applications, and currency conversions must produce consistent results across different system components and external integrations.

The common approach of representing prices as integers in the smallest currency unit (cents for USD) avoids floating-point precision issues but requires consistent application throughout the system. Calculations involving percentages, such as tax rates or discount percentages, must be handled carefully to ensure consistent rounding behavior.

Challenge Category	Specific Problems	Impact on System Design
Inventory Consistency	Race conditions on stock levels, overselling prevention, cart reservation handling	Requires transaction isolation, optimistic/pessimistic locking strategies, inventory validation at multiple stages
Session Management	Cart persistence across devices, session expiry policies, anonymous vs authenticated users	Needs robust session storage, cross-device synchronization, migration from anonymous to authenticated state
Transaction Atomicity	Order creation, payment processing, inventory updates must succeed together	Requires distributed transaction patterns, compensation actions, external system integration strategies
Price Precision	Monetary calculations, tax computation, currency conversion, rounding consistency	Demands integer-based price representation, consistent rounding rules, precision preservation across operations
Concurrent User Actions	Multiple users modifying carts, simultaneous checkout attempts, inventory conflicts	Requires careful concurrency control, queue management, conflict resolution UX patterns
Data Staleness	Product prices changing after cart addition, inventory levels becoming outdated, promotional pricing expiry	Needs cache invalidation strategies, price validation workflows, graceful staleness handling

Existing Approaches Comparison

E-commerce platforms have evolved along several architectural patterns, each addressing the core technical challenges with different trade-offs between complexity, scalability, and development velocity. Understanding these approaches provides context for design decisions throughout the implementation.

Monolithic Architecture represents the traditional approach where all e-commerce functionality resides within a single deployable application. The product catalog, shopping cart management, user authentication, and checkout processing all operate within the same codebase and share a common database. This approach mirrors the physical store analogy most closely, where all operations occur within a single location under unified management.

Monolithic e-commerce platforms excel in maintaining data consistency since all operations occur within the same transaction boundary. Shopping cart operations can immediately validate against current inventory levels. Price updates propagate instantly across all system components. Order processing can atomically update all relevant data structures without complex coordination protocols.

However, monolithic architectures face scalability limitations as traffic grows. Different components of the e-commerce system experience different load patterns - product catalog browsing typically sees much higher traffic than checkout processing, but monolithic deployments cannot scale these components independently. Development velocity also suffers as teams must coordinate changes across the entire codebase.

Microservices Architecture decomposes e-commerce functionality into independent services, each responsible for a specific business capability. Separate services handle product catalog management, shopping cart operations, user authentication, order processing, and payment integration. Each service maintains its own database and communicates with others through well-defined APIs.

Microservices architectures enable independent scaling and development of different e-commerce components. The product catalog service can scale to handle high read traffic while the order processing service optimizes for consistency and reliability. Different teams can develop and deploy services independently, accelerating feature development.

However, microservices introduce distributed system complexity that monolithic architectures avoid. Shopping cart operations may need to validate inventory across service boundaries, introducing network latency and potential failure points. Order processing requires coordination across multiple services, necessitating distributed transaction patterns or eventual consistency models. Data consistency guarantees become weaker and more complex to reason about.

Hybrid Approaches attempt to balance the trade-offs by strategically decomposing certain functionality while maintaining monolithic organization for tightly coupled operations. Common patterns include separating read-heavy catalog browsing from write-heavy transaction processing, or isolating user-facing web applications from administrative back-office systems.

Decision: Monolithic Architecture for Learning Project

- **Context**: This project targets beginner developers learning full-stack development patterns with emphasis on understanding core e-commerce concepts rather than distributed systems complexity.
- **Options Considered**:
  1. Pure monolithic architecture with single database
  2. Microservices architecture with service decomposition
  3. Hybrid approach with selective service separation
- **Decision**: Implement monolithic architecture with clear component boundaries
- **Rationale**: Monolithic architecture eliminates distributed system complexity while still teaching proper component separation, data modeling, and business logic organization. Learners can focus on e-commerce domain concepts without managing service communication, distributed transactions, and deployment orchestration.
- **Consequences**: This approach provides stronger consistency guarantees and simpler debugging but limits scalability learning opportunities and doesn't reflect modern large-scale e-commerce architectures.

Architecture Approach	Advantages	Disadvantages	Best Suited For
Monolithic	Strong consistency, simple deployment, easier debugging, unified transaction management	Limited scalability, single point of failure, slower development velocity at scale	Learning projects, small to medium businesses, rapid prototyping
Microservices	Independent scaling, technology diversity, team autonomy, fault isolation	Distributed system complexity, eventual consistency, network latency, operational overhead	Large scale platforms, multiple development teams, varied scaling requirements
Hybrid	Balanced trade-offs, strategic decomposition, evolution path from monolith	Architectural complexity, decision overhead, potential inconsistency across boundaries	Growing platforms, organizations transitioning architectures, mixed scaling needs

Database Strategy Implications flow directly from architectural choices. Monolithic architectures typically employ a single shared database, enabling ACID transactions across all e-commerce operations. This approach simplifies inventory management, order processing, and data consistency but creates potential bottlenecks as the system scales.

Microservices architectures often adopt database-per-service patterns, where each service maintains its own data store optimized for its specific access patterns. The product catalog service might use a document database optimized for read-heavy workloads, while the order processing service uses a relational database optimized for transactional consistency. This approach enables independent scaling and optimization but complicates cross-service operations and consistency management.

State Management Strategies also vary significantly across architectural approaches. Monolithic systems can maintain session state in server memory, database sessions, or hybrid approaches since all operations occur within the same deployment boundary. Load balancing requires session affinity or shared session storage, but the technical complexity remains manageable.

Distributed architectures must carefully coordinate session state across services. Shopping cart state managed by one service must be accessible to inventory validation services, checkout processing services, and user interface services. This coordination typically requires shared session storage systems, event-driven synchronization patterns, or stateless token-based approaches that embed necessary state information.

The choice between architectural approaches significantly impacts the learning experience and technical complexity that developers must master. Monolithic architectures enable focus on e-commerce domain concepts, business logic implementation,

and user experience design. Microservices architectures shift emphasis toward distributed system patterns, service communication protocols, and operational complexity management.

For this learning-focused project, the monolithic approach provides the optimal balance between realistic e-commerce complexity and manageable technical scope. Learners can master essential patterns like data modeling, business logic organization, and user experience design while avoiding distributed system challenges that can overshadow the core learning objectives.

Implementation Guidance

The technology choices and development setup for this e-commerce platform prioritize learning essential full-stack patterns while maintaining realistic complexity. The recommended approach uses proven technologies with extensive community support and clear learning resources.

Technology Recommendations:

Component	Simple Option	Advanced Option
Backend Framework	Express.js with Node.js (simple routing, middleware)	NestJS (TypeScript, decorators, dependency injection)
Database	SQLite (file-based, no setup required)	PostgreSQL (production-ready, advanced features)
Session Storage	express-session with memory store	Redis with connect-redis for persistence
Authentication	Local username/password with express-session	Passport.js with multiple strategies
Frontend	Vanilla JavaScript with server-rendered HTML	React or Vue.js with API integration
Payment Integration	Mock payment service (no external dependencies)	Stripe API integration (real payment processing)

Recommended Project Structure:

```
ecommerce-store/
├── src/
│   ├── app.js                    ← Express application setup
│   ├── server.js                 ← Server startup and configuration
│   ├── config/
│   │   ├── database.js           ← Database connection setup
│   │   └── session.js            ← Session configuration
│   ├── models/
│   │   ├── User.js               ← User entity and database operations
│   │   ├── Product.js            ← Product entity and catalog operations
│   │   ├── Category.js           ← Product categorization
│   │   ├── Cart.js               ← Shopping cart state management
│   │   └── Order.js              ← Order processing and history
│   ├── routes/
│   │   ├── catalog.js            ← Product browsing and search endpoints
│   │   ├── cart.js               ← Cart management endpoints
│   │   ├── auth.js               ← Authentication endpoints
│   │   └── checkout.js           ← Order processing endpoints
│   ├── middleware/
│   │   ├── auth.js               ← Authentication verification
│   │   └── validation.js         ← Input validation helpers
│   ├── services/
│   │   ├── inventory.js          ← Inventory management business logic
│   │   ├── pricing.js            ← Price calculation and currency handling
│   │   └── payment.js            ← Payment processing abstraction
│   └── utils/
│       ├── validation.js         ← Common validation functions
│       └── currency.js           ← Price formatting and calculation
├── public/
│   ├── css/                      ← Stylesheets
│   ├── js/                       ← Client-side JavaScript
│   └── images/                   ← Product images and assets
├── views/                        ← Server-rendered templates (EJS or Handlebars)
├── tests/
│   ├── unit/                     ← Individual component tests
│   ├── integration/              ← Cross-component tests
│   └── e2e/                      ← Full user journey tests
├── package.json
└── README.md
```

Essential Dependencies Setup:

```javascript
// package.json dependencies for getting started

{

  "dependencies": {

    "express": "^4.18.0",

    "express-session": "^1.17.0",

    "sqlite3": "^5.1.0",

    "bcryptjs": "^2.4.0",

    "express-validator": "^6.14.0",

    "multer": "^1.4.0",

    "ejs": "^3.1.0"

  },

  "devDependencies": {

    "nodemon": "^2.0.0",

    "jest": "^29.0.0",

    "supertest": "^6.3.0"

  }

}
```

Database Setup Helper:

JAVASCRIPT

```javascript
// src/config/database.js - Complete database setup

const sqlite3 = require('sqlite3').verbose();

const path = require('path');


class Database {

    constructor() {

        this.db = null;

    }


    async initialize() {

        const dbPath = path.join(__dirname, '../../ecommerce.db');

        this.db = new sqlite3.Database(dbPath);



        await this.createTables();

        await this.seedInitialData();

    }


    async createTables() {

        const tables = [

            `CREATE TABLE IF NOT EXISTS categories (

                id INTEGER PRIMARY KEY AUTOINCREMENT,

                name TEXT NOT NULL,

                parent_id INTEGER,

                created_at DATETIME DEFAULT CURRENT_TIMESTAMP,

                FOREIGN KEY (parent_id) REFERENCES categories (id)

            )`,



            `CREATE TABLE IF NOT EXISTS products (

                id INTEGER PRIMARY KEY AUTOINCREMENT,

                name TEXT NOT NULL,

                description TEXT,

                price INTEGER NOT NULL,

                inventory_count INTEGER DEFAULT 0,
```

```
            category_id INTEGER,
            image_url TEXT,
            created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (category_id) REFERENCES categories (id)
        )`,

        `CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            email TEXT UNIQUE NOT NULL,
            password_hash TEXT NOT NULL,
            first_name TEXT,
            last_name TEXT,
            created_at DATETIME DEFAULT CURRENT_TIMESTAMP
        )`,

        // Additional tables will be added in subsequent milestones
    ];

    for (const tableSQL of tables) {
        await new Promise((resolve, reject) => {
            this.db.run(tableSQL, (err) => {
                if (err) reject(err);
                else resolve();
            });
        });
    }
}

async seedInitialData() {
    // TODO: Add sample categories and products for development
    // This will be expanded in the Product Catalog milestone
}
```

```
    getConnection() {
        return this.db;
    }
}

module.exports = new Database();
```

Session Configuration:

```javascript
// src/config/session.js - Session management setup
const session = require('express-session');

const sessionConfig = {
    secret: process.env.SESSION_SECRET || 'your-development-secret-key',
    resave: false,
    saveUninitialized: false,
    cookie: {
        secure: process.env.NODE_ENV === 'production',
        httpOnly: true,
        maxAge: 24 * 60 * 60 * 1000 // 24 hours
    }
};

// For production, you would add a proper session store:
// const RedisStore = require('connect-redis')(session);
// sessionConfig.store = new RedisStore({ /* redis config */ });

module.exports = sessionConfig;
```

Development Workflow Setup:

Start with this basic application skeleton that provides the foundation for all milestones:

JAVASCRIPT

```javascript
// src/app.js - Main application setup

const express = require('express');

const session = require('express-session');

const path = require('path');


const database = require('./config/database');

const sessionConfig = require('./config/session');


class EcommerceApp {

    constructor() {

        this.app = express();

        this.setupMiddleware();

        this.setupRoutes();

    }


    setupMiddleware() {

        // TODO: Add body parsing middleware

        // TODO: Add session middleware

        // TODO: Add static file serving

        // TODO: Add authentication middleware

    }


    setupRoutes() {

        // TODO: Mount route handlers for each component

        // Routes will be added as each milestone is completed

    }


    async start(port = 3000) {

        await database.initialize();



        this.app.listen(port, () => {

            console.log(`E-commerce server running on port ${port}`);

        });

    }
```

```
}

module.exports = EcommerceApp;
```

Development Commands:

Create these npm scripts for common development tasks:

```JSON
{
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js",
    "test": "jest",
    "test:watch": "jest --watch",
    "db:reset": "rm -f ecommerce.db && node scripts/setup-database.js",
    "db:seed": "node scripts/seed-data.js"
  }
}
```

Milestone Development Approach:

Each milestone builds incrementally on this foundation:

1. **Milestone 1 (Product Catalog)**: Implement the `Product` and `Category` models, create catalog routes, build product listing and detail pages
2. **Milestone 2 (Shopping Cart)**: Add `Cart` and `CartItem` models, implement cart routes, add session-based cart persistence
3. **Milestone 3 (User Authentication)**: Complete the `User` model, add authentication routes, implement login/logout functionality
4. **Milestone 4 (Checkout Process)**: Add `Order` and `OrderItem` models, implement checkout routes, integrate all previous components

Common Development Pitfalls:

⚠️ **Pitfall: Representing Prices as Floating Point Numbers** JavaScript's floating-point arithmetic can cause precision errors in monetary calculations. Always store prices as integers representing the smallest currency unit (cents for USD). Use helper functions to convert between display format (dollars) and storage format (cents).

⚠️ **Pitfall: Not Validating Input Data** E-commerce systems are attractive targets for malicious input. Validate all user input on both client and server sides. Use libraries like `express-validator` for consistent validation patterns.

⚠️ **Pitfall: Storing Sensitive Data in Sessions** Avoid storing complete user objects or sensitive data in session storage. Store only user IDs and fetch current data from the database when needed. This prevents stale data issues and reduces security exposure.

⚠️ **Pitfall: Not Handling Database Connection Errors** Database connections can fail or become unavailable. Implement proper error handling and connection retry logic. Consider connection pooling for production deployments.

This foundation provides a solid starting point while maintaining flexibility for learners to implement core e-commerce concepts in subsequent milestones. The modular structure supports gradual complexity introduction while following industry-standard

organization patterns.

Goals and Non-Goals

> **Milestone(s):** Foundation for all milestones - establishes scope and constraints that guide implementation decisions

Building an e-commerce platform involves countless potential features, from basic product browsing to sophisticated recommendation engines, multi-vendor marketplaces, and international payment processing. For a learning project focused on mastering full-stack development patterns, it's crucial to establish clear boundaries around what we're building and what we're deliberately omitting. This section serves as the project's charter, defining the functional scope, performance expectations, and explicit exclusions that will guide all subsequent architectural decisions.

Think of this as drawing the blueprint boundaries for a house. Before an architect designs the foundation, room layouts, and electrical systems, they need to know whether they're building a modest starter home or a luxury mansion, whether it needs to accommodate two people or twelve, and whether the budget allows for premium materials or requires cost-effective alternatives. Similarly, our e-commerce platform needs clearly defined limits to ensure we build the right system for our learning objectives without scope creep derailing the educational goals.

The goals outlined here directly correspond to the four major milestones: product catalog functionality, shopping cart management, user authentication, and checkout processing. Each functional goal maps to specific technical challenges that will teach core full-stack development concepts, while the non-goals ensure we maintain focus on fundamental patterns rather than getting lost in peripheral complexities.

Functional Goals

The core functional goals define the essential user capabilities our e-commerce system must provide. These represent the minimum viable product (MVP) that demonstrates a complete e-commerce transaction flow from product discovery through order completion. Each goal corresponds directly to one or more project milestones and introduces specific technical learning opportunities.

Product Discovery and Catalog Management forms the foundation of the user experience. The system must enable customers to discover products through multiple pathways: browsing categorized listings, searching by keywords, and filtering by attributes. This functionality teaches database querying patterns, pagination strategies for large datasets, and search algorithm implementation. The catalog must support hierarchical category organization, allowing products to be grouped into logical categories and subcategories that customers can navigate intuitively.

Catalog Feature	User Capability	Technical Learning
Product Listing	Browse paginated catalog with sort options	Pagination, query optimization, data presentation
Product Search	Find products by name, description, attributes	Search algorithms, indexing strategies
Category Navigation	Browse hierarchical product categories	Tree data structures, recursive queries
Product Details	View comprehensive product information	Data modeling, image handling
Inventory Display	See real-time product availability	Data consistency, concurrent access

Shopping Cart Management enables customers to collect items for potential purchase while maintaining state across their browsing session. This functionality introduces state management challenges, including session persistence, data synchronization between client and server, and handling price changes over time. The cart must support adding items, updating quantities, removing items, and persisting state across page refreshes and browser sessions.

The cart system must validate all operations against current inventory levels to prevent customers from adding unavailable items. It should calculate subtotals and totals in real-time, applying current product prices while handling edge cases like price changes between adding items and checking out. This teaches session management patterns, client-server state synchronization, and data validation strategies.

User Authentication and Profile Management provides secure access to personalized features while teaching fundamental security concepts. The system must support user registration with email and password validation, secure login with password hashing, and session management that persists authenticated state across requests. This functionality introduces cryptographic concepts, security best practices, and session management patterns essential for any web application.

Authentication Feature	Security Requirement	Learning Objective
User Registration	Email validation, password strength	Input validation, security requirements
Password Storage	bcrypt or argon2 hashing	Cryptographic security, password best practices
Login Flow	Credential verification, session creation	Authentication patterns, session management
Session Management	Secure cookies, expiration handling	Web security, state persistence

Order Processing and Checkout represents the culmination of the e-commerce flow, converting cart contents into completed orders. This process must collect shipping information, validate inventory availability at checkout time, create persistent order records, and update inventory levels atomically. This functionality teaches transaction processing, data consistency requirements, and multi-step workflow management.

The checkout process introduces complex timing considerations: inventory must be validated and reserved during checkout to prevent overselling, price calculations must use current prices while handling promotional scenarios, and order creation must be atomic to ensure data consistency. These requirements teach database transaction patterns, concurrency control, and error handling strategies.

Non-Functional Goals

Non-functional goals establish the quality attributes and performance characteristics our system must exhibit. For a learning project, these goals balance realistic production requirements with the constraints of a beginner-friendly implementation. The system should demonstrate good architectural practices without requiring advanced performance optimization techniques.

Performance Requirements focus on acceptable response times and throughput for a small to medium-scale deployment. The system should handle typical user interactions (product browsing, cart updates, checkout) with response times under 500 milliseconds for database operations and under 2 seconds for full page loads. These targets are achievable with basic optimization techniques while teaching performance awareness without requiring complex caching or optimization strategies.

The application should support concurrent users equivalent to a small retail operation - handling 50-100 simultaneous users browsing the catalog and 10-20 concurrent checkout operations without significant performance degradation. This scale teaches basic concurrency considerations without requiring advanced load balancing or distributed system techniques.

Reliability and Data Integrity requirements ensure the system behaves predictably under normal operating conditions while teaching error handling patterns. The application should maintain data consistency during normal operations, preventing issues like inventory overselling, cart corruption, or incomplete orders. Error conditions should be handled gracefully with appropriate user feedback rather than application crashes.

Quality Attribute	Target Metric	Learning Focus
Response Time	<500ms database queries, <2s page loads	Performance awareness, query optimization
Concurrent Users	50-100 browsing, 10-20 checkout	Concurrency patterns, session management
Data Consistency	No inventory overselling, atomic orders	Transaction processing, error handling
Error Handling	Graceful degradation, user feedback	Exception patterns, user experience

Development and Deployment Simplicity ensures the system remains accessible to developers learning full-stack patterns. The application should run on a single server with a local database, avoiding distributed system complexities. The codebase should follow clear separation of concerns with well-defined component boundaries, making it easy to understand and modify individual features without affecting others.

The system should support standard development workflows with local development environments that closely mirror production behavior. This teaches deployment patterns and environment management without requiring complex DevOps tooling or container orchestration systems.

Security Baseline establishes fundamental security practices without implementing enterprise-grade security systems. The application must implement secure password storage using industry-standard hashing algorithms, protect against common web vulnerabilities like SQL injection through parameterized queries, and implement basic session security with secure cookie handling.

These security requirements teach essential web application security patterns while avoiding advanced topics like OAuth integration, multi-factor authentication, or advanced threat detection systems that would complicate the learning experience.

Explicit Non-Goals

Explicit non-goals define functionality that we deliberately exclude from the implementation to maintain focus on core learning objectives. These exclusions prevent scope creep and ensure the project remains appropriate for developers learning full-stack development patterns rather than becoming a comprehensive e-commerce platform.

Advanced E-commerce Features that would complicate the core learning experience are explicitly excluded. The system will not implement product reviews and ratings, which would require additional data modeling, content moderation, and user-generated content management. Wish lists and product comparisons introduce additional state management complexity without teaching fundamentally different patterns from the shopping cart functionality.

Advanced inventory management features like back-order processing, pre-orders, or multi-location inventory tracking are excluded as they introduce complex business logic that obscures the fundamental CRUD and state management patterns the project is designed to teach. Product variants (size, color combinations) with complex pricing rules would require sophisticated data modeling that exceeds beginner-level database design concepts.

Excluded Feature Category	Specific Examples	Reason for Exclusion
User-Generated Content	Product reviews, ratings, Q&A	Content moderation complexity
Advanced Catalog	Product variants, bundles, recommendations	Complex data modeling
Marketing Features	Coupons, promotions, loyalty programs	Business logic complexity
Multi-Vendor	Marketplace, seller management	Distributed system patterns

Payment Processing Integration with real payment gateways is excluded to avoid the complexity of payment processor APIs, PCI compliance requirements, and financial transaction security. The system will implement a payment stub that simulates payment processing without actually charging cards or handling sensitive payment data. This allows learners to understand the checkout

flow and order processing patterns without the overhead of payment gateway integration, webhook handling, and financial data security requirements.

Real payment processing introduces numerous complications including webhook verification, payment failure handling, refund processing, and compliance requirements that would significantly expand the project scope beyond core full-stack development patterns. The payment stub approach teaches the architectural integration points while keeping the focus on data flow and transaction processing patterns.

Administrative and Merchant Features are excluded to maintain focus on customer-facing functionality. The system will not include admin dashboards for inventory management, order fulfillment interfaces, or merchant analytics. These features would require additional user role systems, complex UI development, and business intelligence patterns that exceed the scope of learning basic full-stack development.

Inventory management will be simplified to basic stock level tracking without sophisticated reorder points, supplier integration, or inventory forecasting. Order management will focus on order creation and basic status tracking without implementing complex fulfillment workflows, shipping integration, or return processing systems.

Advanced Technical Features that would require sophisticated infrastructure or advanced programming techniques are explicitly excluded. The system will not implement real-time features like live inventory updates or chat support, which would require WebSocket connections and real-time infrastructure. Caching systems, content delivery networks, or advanced performance optimization techniques are excluded to keep the deployment and infrastructure requirements simple.

Search functionality will use basic database queries rather than implementing full-text search engines or external search services. This teaches fundamental filtering and querying patterns without requiring additional infrastructure components or advanced search algorithms.

Technical Complexity	Excluded Implementation	Simplified Alternative
Real-time Features	WebSocket inventory updates	Periodic page refresh
Advanced Search	Elasticsearch, full-text indexing	Database LIKE queries
Caching	Redis, CDN integration	Simple in-memory caching
Analytics	Business intelligence, reporting	Basic order history

Multi-tenancy and Scaling Features are excluded as they introduce architectural complexity that obscures fundamental development patterns. The system will serve a single merchant rather than supporting multiple stores or vendors. This avoids data isolation requirements, tenant management systems, and the additional complexity of multi-tenant database design patterns.

Horizontal scaling features like load balancing, database sharding, or microservices decomposition are excluded to keep the system architecture straightforward. The monolithic architecture approach teaches clear separation of concerns within a single application while avoiding the distributed system challenges of service communication, data consistency across services, and deployment coordination.

These exclusions ensure that developers can focus on mastering essential full-stack development patterns - database design, API development, state management, authentication, and transaction processing - without being overwhelmed by advanced architectural concerns that are better learned after mastering the fundamentals.

Implementation Guidance

The goals and constraints defined above directly influence technology choices and development approach for our e-commerce platform. This guidance translates the functional and non-functional requirements into concrete technical decisions and development practices.

Technology Selection Based on Goals:

Requirement Category	Simple Approach	Technology Choice
Database	Local file-based database	SQLite with migrations
Authentication	Session-based auth	Express sessions with bcrypt
State Management	Server-side sessions	Express session middleware
Payment Processing	Mock implementation	Stub service with simulated responses
File Upload	Local file storage	Node.js fs module with validation
Search	Database queries	SQL LIKE and filtering

Project Structure Reflecting Goals:

```
ecommerce-store/
├── server/
│   ├── models/              ← Data layer (Product, User, Cart, Order)
│   │   ├── Product.js
│   │   ├── User.js
│   │   ├── Cart.js
│   │   └── Order.js
│   ├── routes/              ← API endpoints by functional area
│   │   ├── catalog.js       ← Product listing, search, categories
│   │   ├── cart.js          ← Cart operations
│   │   ├── auth.js          ← Login, registration, sessions
│   │   └── checkout.js      ← Order processing
│   ├── middleware/          ← Cross-cutting concerns
│   │   ├── auth.js          ← Authentication verification
│   │   └── validation.js    ← Input validation
│   ├── database/            ← Database setup and migrations
│   │   ├── Database.js      ← SQLite connection management
│   │   └── migrations/      ← Schema creation scripts
│   └── app.js              ← EcommerceApp main application
├── client/                  ← Frontend implementation
│   ├── pages/              ← Page components
│   ├── components/         ← Reusable UI components
│   └── utils/              ← Client-side utilities
└── package.json
```

Starter Infrastructure Code:

Complete database connection and session management infrastructure that supports all functional goals:

JAVASCRIPT

```javascript
// server/database/Database.js

const sqlite3 = require('sqlite3').verbose();

const bcrypt = require('bcrypt');

const path = require('path');


class Database {

  constructor() {

    this.db = null;

  }


  initialize() {

    const dbPath = process.env.NODE_ENV === 'production'

      ? './ecommerce.db'

      : './ecommerce_dev.db';



    this.db = new sqlite3.Database(dbPath, (err) => {

      if (err) {

        console.error('Database connection failed:', err.message);

        throw err;

      }

      console.log('Connected to SQLite database');

    });


    // Enable foreign key constraints

    this.db.run('PRAGMA foreign_keys = ON');



    return this.createTables();

  }


  createTables() {

    const tables = [

      // Users table supporting authentication goals

      `CREATE TABLE IF NOT EXISTS users (

        id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
    email TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    first_name TEXT,
    last_name TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
  )`,

  // Categories table supporting hierarchical organization
  `CREATE TABLE IF NOT EXISTS categories (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    parent_id INTEGER,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (parent_id) REFERENCES categories(id)
  )`,

  // Products table with inventory tracking
  `CREATE TABLE IF NOT EXISTS products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    description TEXT,
    price_cents INTEGER NOT NULL,
    inventory_count INTEGER NOT NULL DEFAULT 0,
    category_id INTEGER,
    image_url TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (category_id) REFERENCES categories(id)
  )`,

  // Cart items supporting session-based carts
  `CREATE TABLE IF NOT EXISTS cart_items (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
      session_id TEXT NOT NULL,
      user_id INTEGER,
      product_id INTEGER NOT NULL,
      quantity INTEGER NOT NULL DEFAULT 1,
      created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
      FOREIGN KEY (user_id) REFERENCES users(id),
      FOREIGN KEY (product_id) REFERENCES products(id)
    )`,

    // Orders table for completed purchases
    `CREATE TABLE IF NOT EXISTS orders (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      user_id INTEGER,
      total_cents INTEGER NOT NULL,
      shipping_address TEXT NOT NULL,
      status TEXT DEFAULT 'pending',
      created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
      FOREIGN KEY (user_id) REFERENCES users(id)
    )`,

    // Order items linking orders to products
    `CREATE TABLE IF NOT EXISTS order_items (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      order_id INTEGER NOT NULL,
      product_id INTEGER NOT NULL,
      quantity INTEGER NOT NULL,
      price_cents INTEGER NOT NULL,
      FOREIGN KEY (order_id) REFERENCES orders(id),
      FOREIGN KEY (product_id) REFERENCES products(id)
    )`
  ];
```

```
    return new Promise((resolve, reject) => {
      let completed = 0;
      tables.forEach(sql => {
        this.db.run(sql, (err) => {
          if (err) {
            reject(err);
            return;
          }
          completed++;
          if (completed === tables.length) {
            resolve();
          }
        });
      });
    });
  }

  seedInitialData() {
    // Sample categories and products for development
    const categories = [
      { name: 'Electronics', parent_id: null },
      { name: 'Clothing', parent_id: null }
    ];

    const products = [
      {
        name: 'Laptop',
        description: 'High-performance laptop for work and gaming',
        price_cents: 99999,
        inventory_count: 10,
        category_id: 1
      }
```

```
    ];

    // Implementation would insert sample data for development
  }
}

module.exports = Database;
```

Application Bootstrap Code:

JAVASCRIPT

```javascript
// server/app.js

const express = require('express');

const session = require('express-session');

const SQLiteStore = require('connect-sqlite3')(session);

const Database = require('./database/Database');


class EcommerceApp {

  constructor() {

    this.app = express();

    this.database = new Database();

  }


  setupMiddleware() {

    // Parse JSON bodies

    this.app.use(express.json());



    // Session management supporting cart persistence and auth

    this.app.use(session({

      store: new SQLiteStore({ db: 'sessions.db' }),

      secret: process.env.SESSION_SECRET || 'your-development-secret-key',

      resave: false,

      saveUninitialized: false,

      cookie: {

        maxAge: 24 * 60 * 60 * 1000, // 24 hours

        secure: process.env.NODE_ENV === 'production'

      }

    }));


    // CORS for development

    if (process.env.NODE_ENV !== 'production') {

      this.app.use((req, res, next) => {

        res.header('Access-Control-Allow-Origin', 'http://localhost:3000');

        res.header('Access-Control-Allow-Credentials', true);
```

```
        res.header('Access-Control-Allow-Headers', 'Content-Type');
        next();
      });
    }
  }

  setupRoutes() {
    // Route mounting will be implemented per milestone
    // this.app.use('/api/catalog', require('./routes/catalog'));
    // this.app.use('/api/cart', require('./routes/cart'));
    // this.app.use('/api/auth', require('./routes/auth'));
    // this.app.use('/api/checkout', require('./routes/checkout'));
  }

  async start(port = 3001) {
    await this.database.initialize();
    this.setupMiddleware();
    this.setupRoutes();

    this.app.listen(port, () => {
      console.log(`E-commerce server running on port ${port}`);
    });
  }
}

module.exports = EcommerceApp;
```

Core Logic Skeletons for Each Goal Area:

JAVASCRIPT

```javascript
// server/models/Product.js - Supporting catalog functionality

class Product {

  // TODO: Implement findAll method with pagination and filtering

  // Should support: page, limit, category_id, search_term parameters

  // Return: { products: [], total_count: number, page: number }

  static async findAll(filters = {}) {

    // TODO 1: Build base SQL query with JOIN to categories

    // TODO 2: Add WHERE clauses for category_id and search_term

    // TODO 3: Add LIMIT and OFFSET for pagination

    // TODO 4: Execute count query for total results

    // TODO 5: Execute main query and return structured result

  }


  // TODO: Implement inventory checking for cart operations

  static async checkInventory(productId, requestedQuantity) {

    // TODO 1: Query current inventory_count for product

    // TODO 2: Compare with requested quantity

    // TODO 3: Return { available: boolean, current_stock: number }

  }

}
```

```javascript
// server/models/Cart.js - Supporting cart functionality

class Cart {

  // TODO: Add item to cart with inventory validation

  static async addItem(sessionId, productId, quantity) {

    // TODO 1: Check product inventory availability

    // TODO 2: Check if item already exists in cart

    // TODO 3: If exists, update quantity; if new, insert record

    // TODO 4: Return updated cart totals

  }


  // TODO: Get cart contents with product details

  static async getCart(sessionId) {

    // TODO 1: Query cart_items with JOIN to products

    // TODO 2: Calculate subtotals for each item

    // TODO 3: Calculate cart total

    // TODO 4: Return structured cart data

  }

}
```

Milestone Validation Checkpoints:

After implementing each functional goal area, verify the following behavior:

Milestone 1 (Product Catalog) Checkpoint:

- Start server with `node server/app.js`
- Navigate to product listing page - should show paginated products
- Test category filtering - products should filter by selected category
- Test search functionality - products should filter by search terms
- Verify pagination - should show correct page numbers and navigation

Milestone 2 (Shopping Cart) Checkpoint:

- Add items to cart - should persist across page refreshes
- Update quantities - should recalculate totals correctly
- Remove items - should update cart state immediately
- Test inventory limits - should prevent adding more than available stock

Milestone 3 (Authentication) Checkpoint:

- Register new account - should hash password and create session
- Login with credentials - should authenticate and maintain session
- Access protected routes - should redirect unauthenticated users

- Logout - should clear session and redirect to login

Milestone 4 (Checkout) Checkpoint:

- Complete checkout flow - should create order and clear cart
- Verify inventory updates - should decrement stock after purchase
- Test order confirmation - should display order details and number
- Verify cart persistence - cart should be empty after successful order

Common Implementation Pitfalls:

⚠️ **Pitfall: Price Storage as Floats** Storing prices as floating-point numbers leads to precision errors in calculations. Use integer cents (price_cents) instead of decimal dollars to ensure exact monetary arithmetic.

⚠️ **Pitfall: Missing Session Configuration** Forgetting to configure session store properly results in sessions not persisting across server restarts. Use SQLite session store for development and Redis for production.

⚠️ **Pitfall: Inventory Race Conditions** Checking inventory availability without proper locking can lead to overselling when multiple users purchase simultaneously. Use database transactions for inventory updates.

⚠️ **Pitfall: Password Security** Storing plain text passwords or using weak hashing algorithms compromises user security. Always use bcrypt with appropriate salt rounds (10-12 for development).

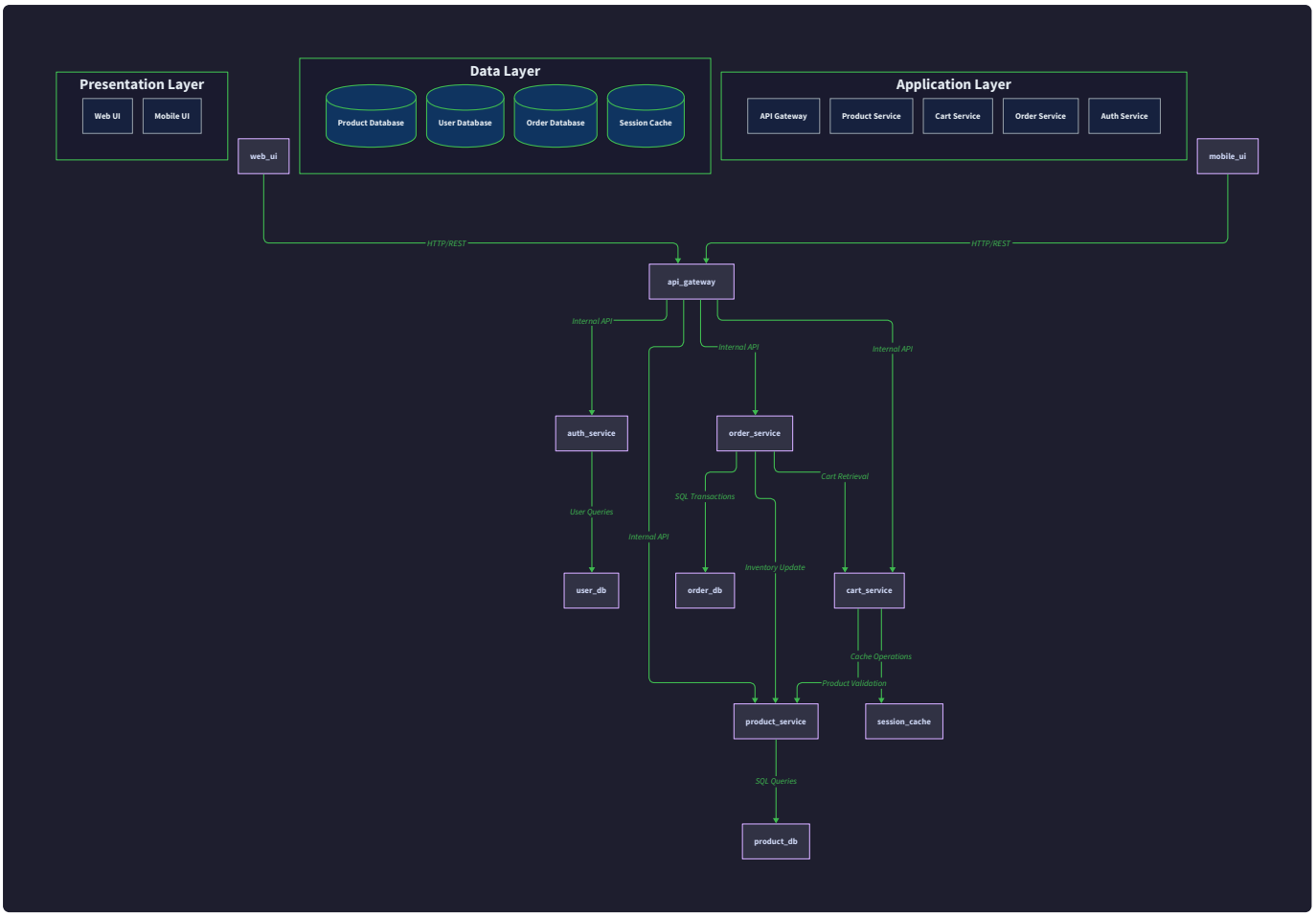
High-Level Architecture

> **Milestone(s):** Foundation for all milestones - provides the structural framework that supports product catalog, shopping cart, user authentication, and checkout process components

Architectural Overview

Think of an e-commerce system like a well-organized department store with three distinct operational levels. The **presentation layer** is like the storefront - beautiful window displays, clear signage, and intuitive navigation that customers interact with directly. The **application layer** is like the store management office - processing customer requests, managing inventory, handling transactions, and enforcing business rules. The **data layer** is like the warehouse - systematically storing products, customer information, orders, and maintaining accurate records of everything that happens in the business.

Our e-commerce platform follows this classic **three-tier web architecture** pattern because it provides clear separation of concerns while remaining simple enough for learning full-stack development. Each tier has distinct responsibilities and communicates through well-defined interfaces, making the system easier to understand, test, and maintain.



The **presentation layer** consists of web browser clients that render HTML pages and execute JavaScript for interactive functionality. This layer handles user interface concerns like form validation, dynamic content updates, and responsive design. It communicates with the application layer exclusively through HTTP requests to REST API endpoints, following the principle of stateless client-server interaction.

The **application layer** is implemented as a Node.js/Express server that exposes REST API endpoints for all business operations. This layer contains the core business logic for product catalog management, shopping cart operations, user authentication, and order processing. It validates incoming requests, enforces business rules, orchestrates database operations, and manages user sessions. The application layer serves as the central coordinator that maintains data consistency and implements the e-commerce workflow.

The **data layer** uses SQLite as the relational database management system, chosen for its simplicity and zero-configuration setup ideal for learning projects. This layer persistently stores all application state including products, users, shopping carts, and orders. SQLite provides ACID transaction guarantees that ensure data consistency during concurrent operations like inventory updates and order processing.

Decision: Three-Tier Architecture

- **Context**: Need to organize a full-stack e-commerce system with frontend, backend, and database components while maintaining clear boundaries and testability
- **Options Considered**: Single-page application with embedded database, microservices architecture, three-tier monolithic architecture
- **Decision**: Three-tier monolithic architecture with presentation, application, and data layers
- **Rationale**: Provides clear separation of concerns without the complexity of distributed systems, matches the learning objectives for full-stack development, and allows independent testing of each layer
- **Consequences**: Enables parallel development of frontend and backend, simplifies deployment and debugging, but requires careful API design to prevent tight coupling between layers

Architecture Option	Pros	Cons	Chosen?
Single-tier (embedded database)	Simple deployment, no network latency	Cannot scale components independently, limited concurrency	No
Three-tier monolithic	Clear separation, easier debugging, simpler deployment	Single failure point, must scale entire application	Yes
Microservices	Independent scaling, technology diversity	Complex deployment, distributed system challenges	No

Component Responsibilities

Each component in our architecture has clearly defined responsibilities that align with the separation of concerns principle. Understanding these boundaries is crucial for maintaining system integrity and enabling independent development of features.

The **EcommerceApp** component serves as the application's main entry point and orchestrator. It initializes the Express server, configures middleware for session management and request parsing, mounts API routes for each business domain, and coordinates startup procedures. This component owns the HTTP server lifecycle and ensures all subsystems are properly initialized before accepting requests.

Component	Primary Responsibility	Secondary Responsibilities
EcommerceApp	HTTP server lifecycle and request routing	Middleware configuration, startup coordination, graceful shutdown
Database	Data persistence and schema management	Connection pooling, transaction management, query optimization
Product Catalog	Product discovery and browsing	Search indexing, category management, price calculations
Shopping Cart	Cart state management and item operations	Session persistence, inventory validation, price synchronization
User Authentication	Identity verification and session management	Password security, session expiration, authorization checks
Checkout Process	Order creation and payment coordination	Address validation, inventory reservation, order confirmation

The **Database** component encapsulates all data access concerns and provides a clean interface for business logic components. It manages the SQLite connection, creates and maintains table schemas, handles transaction boundaries, and provides methods for common database operations. This component ensures data consistency through proper transaction management and prevents SQL injection through parameterized queries.

The **Product Catalog** component handles all product-related operations including listing, searching, filtering, and detailed product information retrieval. It manages the product taxonomy through categories and subcategories, implements search algorithms that match products by name and description, and handles pagination for large product collections. This component also manages product pricing and inventory levels, ensuring accurate information is displayed to customers.

The **Shopping Cart** component manages the stateful shopping experience for both authenticated and anonymous users. It handles adding and removing items, updating quantities with inventory validation, persisting cart state across browser sessions, and maintaining price accuracy as products are added over time. This component bridges the gap between browsing (stateless) and purchasing (transactional) phases of the customer journey.

The **User Authentication** component provides secure identity management through registration, login, logout, and session validation operations. It implements password hashing using bcrypt, manages session cookies with appropriate security settings, and provides middleware for protecting authenticated endpoints. This component ensures that sensitive operations like checkout are only available to verified users.

The **Checkout Process** component orchestrates the complex workflow of converting a shopping cart into a confirmed order. It collects shipping addresses, validates inventory availability, creates order records, updates inventory levels, and coordinates with payment systems. This component must handle partial failures gracefully and maintain transactional integrity throughout the multi-step checkout process.

> The key architectural insight is that each component owns a specific slice of the business domain and communicates with other components through well-defined interfaces. This prevents the common anti-pattern of "God objects" that try to do everything and become impossible to maintain.

Inter-component Communication Patterns:

Components communicate through three primary mechanisms that maintain loose coupling while ensuring reliable data flow:

1. **Direct Method Calls**: Components within the same process communicate through JavaScript method calls with defined interfaces. For example, the Checkout Process component calls `cart.getItems(sessionId)` to retrieve cart contents during order creation.
2. **Database-Mediated Communication**: Components share state through database entities rather than maintaining in-memory references to each other. The Shopping Cart component updates inventory availability through the database, which the Product Catalog component reads during product display.
3. **Event-Driven Updates**: Certain operations trigger cascading updates through event-like patterns. When an order is created, the Checkout Process component updates inventory levels, clears the shopping cart, and creates audit records in a coordinated sequence.

Recommended File Structure

The file structure organizes code into logical modules that mirror the component architecture, making it easy for developers to locate functionality and understand system boundaries. This structure scales well as the application grows and supports both feature development and maintenance activities.

```
ecommerce-store/
├── package.json                    # Project dependencies and scripts
├── server.js                       # Application entry point and startup
├── config/
│   ├── database.js                 # Database connection and configuration
│   └── session.js                  # Session middleware configuration
├── models/
│   ├── database.js                 # Database singleton and table creation
│   ├── product.js                  # Product entity and catalog operations
│   ├── user.js                     # User entity and authentication operations
│   ├── cart.js                     # Cart entity and shopping cart operations
│   └── order.js                    # Order entity and checkout operations
├── routes/
│   ├── products.js                 # Product catalog API endpoints
│   ├── auth.js                     # Authentication API endpoints
│   ├── cart.js                     # Shopping cart API endpoints
│   └── checkout.js                 # Checkout process API endpoints
├── middleware/
│   ├── auth.js                     # Authentication middleware and guards
│   └── validation.js               # Request validation middleware
├── utils/
│   ├── password.js                 # Password hashing utilities
│   ├── validation.js               # Input validation helpers
│   └── errors.js                   # Error handling utilities
├── public/
│   ├── css/                        # Stylesheet assets
│   ├── js/                         # Client-side JavaScript
│   └── images/                     # Product images and static assets
├── views/
│   ├── layout.ejs                  # Base template layout
│   ├── products/                   # Product-related templates
│   ├── cart/                       # Shopping cart templates
│   ├── auth/                       # Authentication templates
│   └── checkout/                   # Checkout process templates
└── tests/
    ├── models/                     # Unit tests for data models
    ├── routes/                     # Integration tests for API endpoints
    └── utils/                      # Unit tests for utility functions
```

Directory Responsibilities and Design Rationale:

The **models/** directory contains all database-related code and business logic, following the Active Record pattern where each model class represents a database table and encapsulates both data and behavior. This approach keeps database concerns separate from HTTP handling while providing a clean interface for business operations.

The **routes/** directory organizes API endpoints by business domain, with each file handling a specific area of functionality. This modular approach makes it easy to locate endpoint definitions and supports parallel development of different features. Each route file focuses on HTTP concerns like request parsing, response formatting, and status code handling.

The **middleware/** directory contains reusable request processing components that implement cross-cutting concerns like authentication, validation, and error handling. Middleware functions follow the Express pattern of receiving request, response, and next parameters, allowing them to be composed into processing pipelines.

The **utils/** directory provides pure functions and utilities that don't fit into the model or route categories. These functions handle common operations like password hashing, input validation, and error formatting. Keeping utilities separate makes them easy to unit test and reuse across different parts of the application.

> **Decision: Domain-Oriented File Structure**
>
> - **Context**: Need to organize code in a way that supports both feature development and maintenance while remaining intuitive for learning projects
> - **Options Considered**: Feature-based organization, layer-based organization, domain-oriented organization
> - **Decision**: Domain-oriented organization with separation of models, routes, and utilities
> - **Rationale**: Mirrors the mental model of business domains (products, users, carts, orders), makes it easy to locate related functionality, and supports the MVC pattern familiar to web developers
> - **Consequences**: Related functionality is co-located, testing is straightforward, but some cross-cutting concerns require careful organization to avoid duplication

Module Dependencies and Loading Order:

The application follows a specific initialization sequence that ensures dependencies are available when needed:

1. **Configuration Loading**: The `config/` modules are loaded first to establish database connections and session settings before any business logic executes.
2. **Database Initialization**: The `Database` singleton connects to SQLite, creates tables if they don't exist, and seeds initial data for development environments.
3. **Model Registration**: Business logic models are instantiated and configured with their database dependencies, establishing the core domain objects.
4. **Middleware Setup**: Authentication and validation middleware are configured and made available to route handlers.
5. **Route Mounting**: API routes are mounted on the Express application, creating the public interface for client applications.
6. **Server Startup**: The HTTP server begins accepting requests only after all components are successfully initialized.

This initialization order prevents common startup issues like undefined dependencies or missing database tables, while providing clear error messages when configuration problems occur.

Common File Structure Pitfalls:

⚠️ **Pitfall: Circular Dependencies** Many beginners create circular import patterns where models import routes and routes import models, causing Node.js module loading failures. This happens when trying to share validation logic between models and routes. The fix is to extract shared validation logic into the `utils/` directory and import it into both models and routes.

⚠️ **Pitfall: Giant Single Files** Another common mistake is putting all routes in a single `routes.js` file or all models in one `models.js` file. As the application grows, these files become impossible to navigate and multiple developers cannot work on them simultaneously. Split files by business domain from the beginning, even if they start small.

⚠️ **Pitfall: Database Code in Routes** Beginners often write SQL queries directly in route handlers, mixing HTTP concerns with data access. This makes testing difficult and violates separation of concerns. Always encapsulate database operations in model classes and call model methods from routes.

⚠️ **Pitfall: Missing Error Boundaries** Without proper error handling organization, exceptions bubble up unpredictably and crash the application. Create a centralized error handling utility in `utils/errors.js` and use it consistently across all components to provide user-friendly error responses.

Implementation Guidance

The implementation guidance provides practical direction for translating the architectural design into working code, with specific recommendations for technology choices and file organization that support the learning objectives.

Technology Recommendations:

Component	Simple Option	Advanced Option
Web Framework	Express.js with EJS templates	Express.js with React frontend
Database	SQLite with sqlite3 package	PostgreSQL with Sequelize ORM
Session Management	express-session with memory store	express-session with Redis store
Authentication	bcrypt with sessions	JWT tokens with refresh mechanism
File Upload	multer for local filesystem	multer with AWS S3 integration
Testing	Jest with supertest	Jest + Cypress for E2E testing

Core Application Structure:

The main application entry point establishes the foundation for all other components:

JAVASCRIPT

```javascript
// server.js - Application entry point

const express = require('express');

const session = require('express-session');

const path = require('path');

const { Database } = require('./models/database');

const authRoutes = require('./routes/auth');

const productRoutes = require('./routes/products');

const cartRoutes = require('./routes/cart');

const checkoutRoutes = require('./routes/checkout');


class EcommerceApp {

    constructor() {

        this.app = express();

        this.database = null;

    }


    async initialize() {

        // TODO: Initialize database connection and create tables

        // TODO: Setup middleware for sessions, parsing, and static files

        // TODO: Configure view engine for server-side rendering

        // TODO: Mount API routes for each business domain

        // Hint: Order matters - setup middleware before routes

    }


    setupMiddleware() {

        // TODO: Configure express-session with SESSION_SECRET

        // TODO: Setup express.json() and express.urlencoded() parsers

        // TODO: Configure static file serving for CSS/JS/images

        // TODO: Setup EJS view engine with views directory

        // Hint: Session maxAge should be 24 hours for good UX

    }


    setupRoutes() {

        // TODO: Mount /api/auth routes for authentication
```

```
        // TODO: Mount /api/products routes for catalog
        // TODO: Mount /api/cart routes for shopping cart
        // TODO: Mount /api/checkout routes for order processing
        // TODO: Setup catch-all route for frontend navigation
    }

    async start(port = 3000) {
        // TODO: Initialize database and seed initial data
        // TODO: Setup middleware and routes
        // TODO: Start HTTP server and log startup message
        // Hint: Use process.env.NODE_ENV to detect development mode
    }
}

// Application startup
const app = new EcommerceApp();
app.start(process.env.PORT || 3000);
```

Database Foundation:

The database component provides the persistence layer foundation:

JAVASCRIPT

```javascript
// models/database.js - Database singleton and schema management

const sqlite3 = require('sqlite3').verbose();

const path = require('path');


class Database {

    constructor() {

        this.db = null;

    }


    async initialize() {

        // TODO: Create SQLite database connection

        // TODO: Enable foreign key constraints

        // TODO: Create all application tables

        // TODO: Seed initial data in development mode

        // Hint: Use path.join(__dirname, '../data/store.db') for database file

    }


    async createTables() {

        // TODO: Create users table with email, password_hash, created_at

        // TODO: Create categories table with name, parent_id for hierarchy

        // TODO: Create products table with name, price, inventory, category_id

        // TODO: Create carts table with session_id, created_at, updated_at

        // TODO: Create cart_items table linking carts to products with quantities

        // TODO: Create orders table with user_id, total, status, shipping_address

        // TODO: Create order_items table linking orders to products with quantities

        // Hint: Use INTEGER PRIMARY KEY AUTOINCREMENT for ID fields

    }


    async seedInitialData() {

        // TODO: Insert sample categories (Electronics, Clothing, Books)

        // TODO: Insert sample products with realistic prices and inventory

        // TODO: Only seed if NODE_ENV is 'development'

        // Hint: Check if data already exists before inserting

    }
```

```
    getConnection() {
        return this.db;
    }
}

module.exports = { Database: new Database() };
```

Session and Authentication Configuration:

```javascript
// config/session.js - Session configuration
const session = require('express-session');

const SESSION_SECRET = process.env.SESSION_SECRET || 'dev-secret-change-in-production';
const NODE_ENV = process.env.NODE_ENV || 'development';

const sessionConfig = {
    secret: SESSION_SECRET,
    resave: false,
    saveUninitialized: false,
    cookie: {
        maxAge: 24 * 60 * 60 * 1000, // 24 hours
        httpOnly: true,
        secure: NODE_ENV === 'production', // HTTPS only in production
        sameSite: 'lax'
    }
};

module.exports = sessionConfig;
```

Model Base Structure:

Each business domain gets its own model file with consistent patterns:

JAVASCRIPT

```javascript
// models/product.js - Product catalog operations

const { Database } = require('./database');

class Product {
    static async findAll(options = {}) {
        // TODO: Implement product listing with pagination
        // TODO: Support category filtering if options.category provided
        // TODO: Support search if options.search provided
        // TODO: Support sorting by price, name, or created_at
        // TODO: Return { products: [], totalCount: number, page: number }
        // Hint: Use LIMIT and OFFSET for pagination
    }

    static async findById(id) {
        // TODO: Find single product by ID
        // TODO: Include category information via JOIN
        // TODO: Return null if product not found
        // Hint: Use parameterized queries to prevent SQL injection
    }

    static async search(query, options = {}) {
        // TODO: Search products by name and description
        // TODO: Use LIKE operator with wildcards for flexible matching
        // TODO: Apply same pagination and filtering as findAll
        // Hint: Search should be case-insensitive
    }

    static async updateInventory(productId, quantityChange) {
        // TODO: Update product inventory by adding quantityChange
        // TODO: Ensure inventory cannot go below zero
        // TODO: Use database transaction for consistency
        // TODO: Return updated inventory level or throw error
        // Hint: This is called during checkout to decrement stock
    }
```

```
}

module.exports = Product;
```

API Route Structure:

Routes handle HTTP concerns and delegate business logic to models:

```javascript
// routes/products.js - Product catalog API endpoints
const express = require('express');
const Product = require('../models/product');
const router = express.Router();

// GET /api/products - List products with filtering and pagination
router.get('/', async (req, res) => {
    // TODO: Extract query parameters (page, category, search, sort)
    // TODO: Validate parameters and set defaults
    // TODO: Call Product.findAll() with options
    // TODO: Return JSON response with products and pagination info
    // TODO: Handle errors and return appropriate status codes
    // Hint: Use parseInt() to convert string parameters to numbers
});

// GET /api/products/:id - Get single product details
router.get('/:id', async (req, res) => {
    // TODO: Extract product ID from request params
    // TODO: Validate ID is a positive integer
    // TODO: Call Product.findById() to get product data
    // TODO: Return 404 if product not found, 200 with product data if found
    // TODO: Handle database errors with 500 status
});

module.exports = router;
```

Language-Specific Implementation Hints:

For Node.js/Express development, follow these specific practices:

- **Async/Await Pattern**: Use `async/await` consistently instead of callbacks for database operations. This makes error handling more predictable and code easier to read.
- **Parameter Validation**: Use `parseInt()` and `parseFloat()` to convert string parameters to numbers, and always validate the results aren't `NaN`. Express parameters are always strings by default.
- **SQL Parameterization**: Always use parameterized queries with the `sqlite3` package: `db.get('SELECT * FROM products WHERE id = ?', [id], callback)` to prevent SQL injection.
- **Error Handling**: Wrap database operations in try-catch blocks and return appropriate HTTP status codes (400 for client errors, 500 for server errors, 404 for not found).
- **Session Access**: Access session data through `req.session` in route handlers. Create session properties directly: `req.session.userId = user.id` for login state.

Milestone Checkpoint:

After implementing the high-level architecture, verify the foundation is working correctly:

1. **Start the application**: Run `node server.js` and confirm the server starts without errors and logs the listening port.
2. **Database initialization**: Check that the SQLite database file is created in the correct location and contains all required tables with proper schema.
3. **Route mounting**: Test that basic routes respond correctly - `curl http://localhost:3000/api/products` should return product data (even if empty initially).
4. **Session functionality**: Visit the application in a browser and confirm that session cookies are set correctly using browser developer tools.
5. **Static file serving**: Confirm that CSS, JavaScript, and image files are served correctly from the `public/` directory.

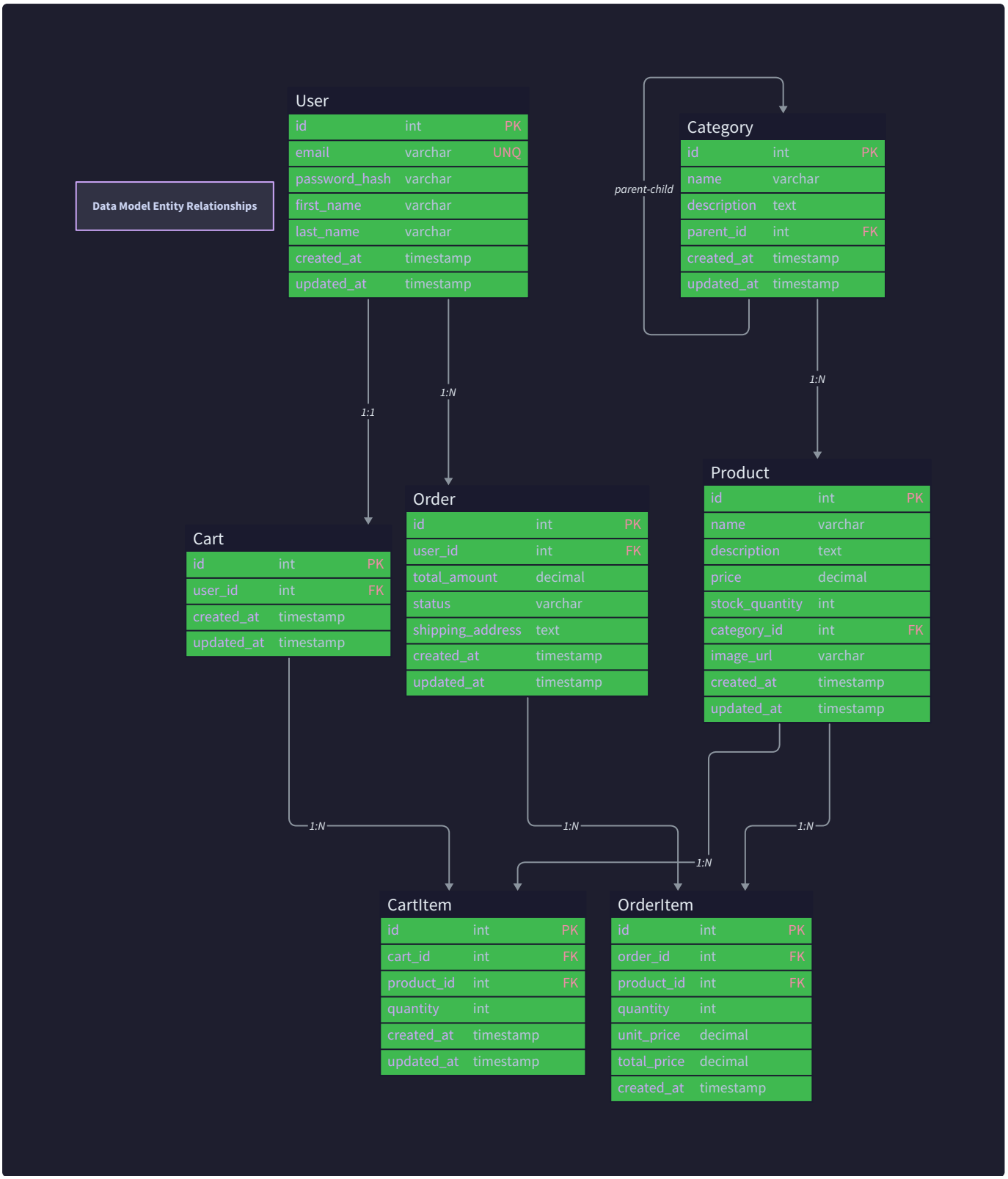
Expected behavior: The application should start successfully, create database tables, accept HTTP requests, and serve both API endpoints and static files. Error messages should be clear and point to specific configuration issues if anything is misconfigured.

Signs of problems: If the server fails to start, check that all required npm packages are installed and that the database directory is writable. If routes return 404 errors, verify that route mounting happens after middleware setup in the correct order.

Data Model

> **Milestone(s):** Foundation for all milestones - provides the persistent storage structure that supports product catalog (Milestone 1), shopping cart (Milestone 2), user authentication (Milestone 3), and checkout process (Milestone 4)

The data model serves as the foundation for our e-commerce platform, defining how we store and relate all the critical business information. Think of the data model as the filing system for a large retail operation - it needs to organize products, track customer information, manage shopping carts, and record completed orders in a way that maintains accuracy and supports fast retrieval. Just as a physical store needs organized shelves, inventory logs, customer records, and sales receipts, our digital store requires structured data entities that capture these same concepts with precision and integrity.



The data model must handle several complex relationships simultaneously. Products belong to categories in a hierarchical structure, users maintain shopping carts that persist across sessions, and completed orders preserve a snapshot of products and prices at the time of purchase. The challenge lies in maintaining data consistency across these relationships while supporting concurrent operations from multiple users browsing products, updating carts, and completing purchases simultaneously.

Core Entities

The e-commerce system revolves around four primary entities that capture the essential business concepts: products available for purchase, users who interact with the system, shopping carts that track intended purchases, and orders that record completed transactions. Each entity serves a specific purpose in the customer journey from product discovery through purchase completion.

Product Entity

The `Product` entity represents individual items available for purchase in the catalog. Think of each product record as a detailed product card in a physical store - it contains all the information a customer needs to make a purchasing decision, plus the operational data needed for inventory management. Products form the core of the catalog browsing experience and serve as the foundation for cart items and order line items.

Field Name	Type	Description
id	INTEGER PRIMARY KEY	Auto-incrementing unique identifier for the product
name	VARCHAR(255) NOT NULL	Human-readable product name displayed in the catalog
description	TEXT	Detailed product description with features and specifications
price	INTEGER NOT NULL	Price in cents to avoid floating-point precision errors
inventory_count	INTEGER DEFAULT 0	Current available stock quantity for purchase
category_id	INTEGER	Foreign key reference to the product's category
image_url	VARCHAR(500)	URL path to the primary product image
sku	VARCHAR(100) UNIQUE	Stock Keeping Unit identifier for inventory tracking
is_active	BOOLEAN DEFAULT TRUE	Flag indicating if product is available for purchase
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Record creation timestamp
updated_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Last modification timestamp

The price field stores monetary values as integers in cents rather than decimal types to prevent floating-point precision errors that can accumulate during calculations. For example, a product priced at $29.99 stores the value 2999. This approach ensures exact monetary calculations throughout the system, preventing discrepancies between displayed prices and charged amounts.

User Entity

The `User` entity represents registered customers who can authenticate with the system and maintain persistent shopping carts. Each user record serves as both an authentication credential store and a customer profile that links to their shopping history. The user entity enables personalized experiences and order tracking across multiple sessions.

Field Name	Type	Description
id	INTEGER PRIMARY KEY	Auto-incrementing unique identifier for the user
email	VARCHAR(255) UNIQUE NOT NULL	Email address used for authentication and communication
password_hash	VARCHAR(255) NOT NULL	Bcrypt hashed password for secure authentication
first_name	VARCHAR(100)	Customer's first name for personalization
last_name	VARCHAR(100)	Customer's last name for shipping and billing
phone	VARCHAR(20)	Contact phone number for order notifications
is_active	BOOLEAN DEFAULT TRUE	Flag indicating if account is enabled
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Account creation timestamp
updated_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Last profile modification timestamp

The password_hash field stores bcrypt-hashed passwords with salt, never storing plaintext passwords. The email field serves as the unique username for authentication while also providing a communication channel for order confirmations and updates. The is_active flag allows for account suspension without data deletion, preserving order history for reactivated accounts.

Category Entity

The `Category` entity organizes products into a hierarchical taxonomy that supports nested subcategories. Think of categories as the department and aisle organization in a physical store - they help customers navigate to relevant products and enable filtered browsing. The hierarchical structure allows for broad categories like "Electronics" with subcategories like "Smartphones" and "Laptops."

Field Name	Type	Description
id	INTEGER PRIMARY KEY	Auto-incrementing unique identifier for the category
name	VARCHAR(255) NOT NULL	Human-readable category name displayed in navigation
description	TEXT	Optional description explaining the category scope
parent_id	INTEGER	Foreign key reference to parent category (NULL for root)
slug	VARCHAR(255) UNIQUE	URL-friendly identifier for category pages
display_order	INTEGER DEFAULT 0	Sort order for category display in navigation
is_active	BOOLEAN DEFAULT TRUE	Flag indicating if category appears in navigation
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Record creation timestamp
updated_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Last modification timestamp

The parent_id field creates the hierarchical structure by referencing another category record. Root categories have a NULL parent_id, while subcategories reference their parent category. This self-referential relationship enables unlimited nesting depth while maintaining simple queries for category trees.

Cart Entity

The `Cart` entity represents a user's current shopping session, tracking items they intend to purchase before checkout. Unlike a physical shopping cart that exists only while in the store, digital carts persist across browser sessions and device changes. Each cart belongs to either an authenticated user or an anonymous session identifier.

Field Name	Type	Description
id	INTEGER PRIMARY KEY	Auto-incrementing unique identifier for the cart
user_id	INTEGER	Foreign key reference to the cart owner (NULL for anonymous)
session_id	VARCHAR(255)	Session identifier for anonymous carts
status	VARCHAR(20) DEFAULT 'active'	Cart status: active, abandoned, or converted
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Cart creation timestamp
updated_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Last cart modification timestamp
expires_at	TIMESTAMP	Expiration timestamp for abandoned cart cleanup

The cart uses either user_id for authenticated users or session_id for anonymous browsing sessions. This dual approach ensures cart persistence regardless of authentication status while enabling anonymous users to maintain carts across page reloads. The status field tracks cart lifecycle states for analytics and cleanup processes.

CartItem Entity

The `CartItem` entity represents individual product selections within a shopping cart, tracking the product, quantity, and price at the time of addition. Each cart item links a specific product to a cart with the customer's desired quantity. The cart item preserves the product price when added to handle price changes between cart addition and checkout.

Field Name	Type	Description
id	INTEGER PRIMARY KEY	Auto-incrementing unique identifier for the cart item
cart_id	INTEGER NOT NULL	Foreign key reference to the containing cart
product_id	INTEGER NOT NULL	Foreign key reference to the selected product
quantity	INTEGER NOT NULL DEFAULT 1	Number of units the customer wants to purchase
unit_price	INTEGER NOT NULL	Price per unit in cents when added to cart
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Item addition timestamp
updated_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Last quantity update timestamp

The unit_price field captures the product price at the time of cart addition, protecting customers from unexpected price increases during their shopping session. The combination of cart_id and product_id should be unique to prevent duplicate entries for the same product in a single cart.

Order Entity

The `Order` entity represents a completed purchase transaction, preserving all details necessary for fulfillment and customer service. Think of an order as a comprehensive receipt that captures not just what was purchased, but also when, by whom, and at what prices. Orders provide an immutable record of the transaction state at completion time.

Field Name	Type	Description
id	INTEGER PRIMARY KEY	Auto-incrementing unique identifier for the order
user_id	INTEGER	Foreign key reference to the purchasing customer
order_number	VARCHAR(50) UNIQUE NOT NULL	Human-readable order identifier for customer reference
status	VARCHAR(20) DEFAULT 'pending'	Order status: pending, confirmed, shipped, delivered, cancelled
subtotal	INTEGER NOT NULL	Sum of all order items in cents before taxes
tax_amount	INTEGER DEFAULT 0	Calculated tax amount in cents
shipping_amount	INTEGER DEFAULT 0	Shipping cost in cents
total_amount	INTEGER NOT NULL	Final order total including taxes and shipping
shipping_first_name	VARCHAR(100) NOT NULL	Recipient first name for shipping address
shipping_last_name	VARCHAR(100) NOT NULL	Recipient last name for shipping address
shipping_address_line1	VARCHAR(255) NOT NULL	Primary shipping address line
shipping_address_line2	VARCHAR(255)	Secondary shipping address line (apartment, suite)
shipping_city	VARCHAR(100) NOT NULL	Shipping city name
shipping_state	VARCHAR(100) NOT NULL	Shipping state or province
shipping_postal_code	VARCHAR(20) NOT NULL	Shipping postal or ZIP code
shipping_country	VARCHAR(100) DEFAULT 'United States'	Shipping country name
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Order placement timestamp
updated_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Last order modification timestamp

The order_number provides a customer-friendly identifier like "ORD-2023-001234" that customers can reference in support inquiries. All monetary fields store values in cents for precision. The shipping address fields are denormalized within the order to preserve the delivery address even if the customer later updates their profile address.

OrderItem Entity

The `OrderItem` entity represents individual products within a completed order, capturing the exact product details, quantity, and pricing at the time of purchase. Order items serve as line items on the customer's receipt and preserve product information even if the original product record changes or gets deleted from the catalog.

Field Name	Type	Description
id	INTEGER PRIMARY KEY	Auto-incrementing unique identifier for the order item
order_id	INTEGER NOT NULL	Foreign key reference to the containing order
product_id	INTEGER	Foreign key reference to the original product (may be NULL if deleted)
product_name	VARCHAR(255) NOT NULL	Product name at time of purchase
product_description	TEXT	Product description at time of purchase
quantity	INTEGER NOT NULL	Number of units purchased
unit_price	INTEGER NOT NULL	Price per unit in cents at time of purchase
line_total	INTEGER NOT NULL	Total for this line item (quantity × unit_price)
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Order item creation timestamp

Order items denormalize product information to preserve purchase details even if products are later modified or removed from the catalog. This ensures order history remains accurate and complete for customer service, returns processing, and financial reporting.

Entity Relationships

The entity relationships define how data connects across the system to support complex e-commerce operations. Understanding these relationships is crucial for maintaining data integrity and implementing efficient queries. Think of entity relationships as the organizational chart of data dependencies - they define which pieces of information depend on others and how changes propagate through the system.

User-Centric Relationships

Users serve as the central hub for personalized data, connecting to carts and orders through direct foreign key relationships. Each user can have multiple carts over time (though typically only one active cart) and multiple orders representing their purchase history. This one-to-many relationship enables customer profile management and order tracking functionality.

The User-to-Cart relationship supports both authenticated and anonymous scenarios. Authenticated users link carts through the user_id foreign key, while anonymous users rely on session_id tracking. When anonymous users authenticate, the system can transfer their session-based cart to their user account, preserving their shopping progress.

The User-to-Order relationship maintains a complete purchase history for each customer. Orders retain the user_id reference even if the user account is later deactivated, ensuring financial records and order fulfillment data remain intact for business operations and compliance requirements.

Product-Category Hierarchy

Products connect to the category hierarchy through the category_id foreign key, establishing which category contains each product. This relationship enables category-based browsing and filtering, allowing customers to explore products within specific departments or subcategories.

Categories form a self-referential hierarchy through the parent_id field, creating a tree structure of departments and subdepartments. Root categories have parent_id set to NULL, while subcategories reference their parent category's ID. This hierarchical design supports unlimited nesting depth for complex product taxonomies.

The Product-to-Category relationship is many-to-one, meaning each product belongs to exactly one category, but each category can contain multiple products. This simplifies product management while supporting rich category-based navigation and filtering experiences.

Cart-Product Relationships

Shopping carts connect to products through the intermediary CartItem entity, creating a many-to-many relationship that tracks quantities and pricing. Each CartItem links one product to one cart with a specific quantity, enabling customers to add multiple units of the same product or multiple different products to their cart.

The Cart-to-CartItem relationship is one-to-many, where each cart contains zero or more cart items. CartItems connect to Products through product_id foreign keys, preserving the product selection and unit price at the time of cart addition. This design handles price changes gracefully by maintaining the original cart price until checkout.

Cart relationships support both persistent and temporary scenarios. Authenticated user carts persist across sessions through the user_id foreign key, while anonymous carts rely on session_id tracking and eventual cleanup through the expires_at timestamp.

Order-Product Relationships

Completed orders connect to products through OrderItem entities, creating an immutable record of purchase details. The Order-to-OrderItem relationship captures the exact product information, quantities, and pricing at the time of purchase, preserving this data even if the original product catalog changes.

OrderItems maintain both a product_id foreign key (which may become NULL if products are deleted) and denormalized product information including name, description, and pricing. This dual approach preserves referential links when possible while ensuring order history remains complete and accurate regardless of future catalog changes.

The Order-to-OrderItem relationship is one-to-many, where each order contains one or more order items representing the products purchased. OrderItems calculate line_total as quantity × unit_price, with order-level subtotal and total_amount aggregating across all order items plus taxes and shipping.

Data Constraints and Validation

Data constraints enforce business rules at the database level to maintain data integrity and prevent invalid states. These constraints serve as the final safeguard against data corruption, working alongside application-level validation to ensure the system maintains consistent and accurate information. Think of constraints as the safety mechanisms in a manufacturing process - they prevent defective products from entering the system even when other quality checks fail.

Primary Key Constraints

Every entity uses an auto-incrementing integer primary key to ensure unique identification and efficient indexing. Primary keys provide stable references for foreign key relationships and enable predictable query performance. The auto-increment behavior guarantees uniqueness without requiring application-level coordination, supporting concurrent insert operations across multiple users.

Primary key selection impacts system scalability and data distribution. Integer primary keys offer excellent performance for single-database deployments while providing a foundation for future partitioning strategies. The sequential nature of auto-increment keys also optimizes database page utilization and index maintenance operations.

Foreign Key Constraints

Foreign key constraints enforce referential integrity between related entities, preventing orphaned records and maintaining relationship consistency. Each foreign key relationship includes appropriate constraint actions for update and delete operations to handle cascading changes gracefully.

Relationship	Foreign Key	Constraint Action	Rationale
Product → Category	product.category_id	SET NULL ON DELETE	Products can exist without categories for uncategorized items
CartItem → Cart	cartitem.cart_id	CASCADE ON DELETE	Cart items should be removed when carts are deleted
CartItem → Product	cartitem.product_id	RESTRICT ON DELETE	Prevent product deletion while items exist in active carts
OrderItem → Order	orderitem.order_id	CASCADE ON DELETE	Order items are meaningless without their parent order
OrderItem → Product	orderitem.product_id	SET NULL ON DELETE	Preserve order history even if products are later removed
Cart → User	cart.user_id	CASCADE ON DELETE	Remove user carts when accounts are deleted
Order → User	order.user_id	SET NULL ON DELETE	Preserve order records for business reporting

Uniqueness Constraints

Uniqueness constraints prevent duplicate data that could cause business logic errors or customer confusion. These constraints enforce key business rules like unique email addresses for user accounts and unique SKU codes for product identification.

The User.email field enforces unique email addresses to prevent authentication conflicts and ensure reliable customer communication. Product.sku enforces unique stock keeping unit codes for accurate inventory tracking and supplier coordination. Category.slug ensures unique URL paths for category pages to prevent navigation conflicts.

Order.order_number enforces unique customer-facing order identifiers to prevent confusion in customer service and order tracking scenarios. The combination of cart_id and product_id in CartItem should be unique to prevent duplicate product entries within a single shopping cart.

Data Type and Range Constraints

Data type constraints ensure fields contain valid formats and ranges appropriate for their business purpose. Monetary fields use INTEGER types storing cent values to prevent floating-point precision errors. Timestamp fields use appropriate date/time types with timezone awareness for accurate temporal tracking.

String fields include appropriate length limits based on expected content. Email addresses allow up to 255 characters to accommodate longer domain names, while names and descriptions use reasonable limits that support international character sets. URL fields provide sufficient length for image paths and external links.

Business Logic Constraints

Business logic constraints enforce domain-specific rules that maintain system consistency. These constraints prevent invalid business states that could cause operational problems or customer dissatisfaction.

Constraint	Entity.Field	Rule	Enforcement
Positive Pricing	Product.price	price > 0	CHECK constraint
Non-negative Inventory	Product.inventory_count	inventory_count >= 0	CHECK constraint
Positive Quantities	CartItem.quantity	quantity > 0	CHECK constraint
Positive Order Amounts	Order.total_amount	total_amount > 0	CHECK constraint
Valid Email Format	User.email	email contains @ and domain	Application validation
Password Complexity	User.password_hash	minimum length and complexity	Application validation
Future Expiration	Cart.expires_at	expires_at > created_at	Application validation

Decision: Monetary Precision Strategy

- **Context**: E-commerce systems must handle monetary calculations with perfect precision to prevent customer billing discrepancies and financial reporting errors
- **Options Considered**: DECIMAL database type, FLOAT database type, INTEGER cent storage
- **Decision**: Store all monetary values as INTEGER fields representing cents
- **Rationale**: Integer arithmetic avoids floating-point precision errors that can accumulate during tax calculations, discounts, and currency conversions. Storing cents as integers ensures exact calculations and prevents penny discrepancies that damage customer trust
- **Consequences**: All monetary calculations require cent-to-dollar conversion for display, but this ensures mathematical precision throughout the system

Validation Error Handling

Constraint violations generate specific error messages that help developers and support staff identify data integrity issues. Primary key violations indicate duplicate record creation attempts, while foreign key violations suggest referential integrity problems or race conditions in concurrent operations.

Uniqueness constraint violations often indicate user interface issues like duplicate form submissions or inadequate client-side validation. Business logic constraint violations suggest application bugs or malicious input attempts that bypassed client-side validation.

The system should log all constraint violations with sufficient context for debugging while returning user-friendly error messages that guide customers toward resolution. For example, email uniqueness violations should prompt users to use the login flow rather than exposing technical database details.

Implementation Guidance

This section provides concrete technology recommendations and starter code to implement the data model using Node.js and SQLite for development simplicity with migration paths to PostgreSQL for production deployments.

Technology Recommendations

Component	Simple Option	Advanced Option
Database	SQLite with better-sqlite3	PostgreSQL with pg driver
Schema Migration	Direct SQL CREATE statements	Knex.js migration framework
ORM Layer	Raw SQL queries with prepared statements	Sequelize ORM with model definitions
Validation	Manual constraint checking	Joi schema validation library
Connection Pool	Single SQLite connection	PostgreSQL connection pool (pg-pool)

Recommended File Structure

```
project-root/
├── src/
│   ├── models/
│   │   ├── database.js          ← Database connection and schema setup
│   │   ├── product.js           ← Product model with CRUD operations
│   │   ├── user.js              ← User model with authentication helpers
│   │   ├── category.js          ← Category model with hierarchy queries
│   │   ├── cart.js              ← Cart model with session management
│   │   └── order.js             ← Order model with transaction handling
│   ├── migrations/
│   │   ├── 001_initial_schema.sql ← DDL statements for table creation
│   │   └── 002_seed_data.sql    ← Sample data for development
│   └── app.js                   ← Main application entry point
├── database/
│   └── ecommerce.sqlite         ← SQLite database file (development)
└── package.json
```

Database Infrastructure Starter Code

JAVASCRIPT

```javascript
// src/models/database.js

const Database = require('better-sqlite3');

const path = require('path');

const fs = require('fs');


class DatabaseManager {

    constructor() {

        this.db = null;

    }


    /**

     * Initializes database connection and creates tables if they don't exist.

     * Sets up foreign key enforcement and transaction isolation.

     */

    initialize() {

        const dbPath = path.join(__dirname, '../../database/ecommerce.sqlite');



        // Ensure database directory exists

        const dbDir = path.dirname(dbPath);

        if (!fs.existsSync(dbDir)) {

            fs.mkdirSync(dbDir, { recursive: true });

        }


        this.db = new Database(dbPath);



        // Enable foreign key constraints and WAL mode for better concurrency

        this.db.pragma('foreign_keys = ON');

        this.db.pragma('journal_mode = WAL');



        this.createTables();

        return this;

    }


    /**
```

```
     * Creates all database tables with proper constraints and indexes.
     * Executes DDL statements in dependency order to handle foreign keys.
     */
    createTables() {
        // Categories table (no dependencies)
        this.db.exec(`
            CREATE TABLE IF NOT EXISTS categories (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name VARCHAR(255) NOT NULL,
                description TEXT,
                parent_id INTEGER,
                slug VARCHAR(255) UNIQUE NOT NULL,
                display_order INTEGER DEFAULT 0,
                is_active BOOLEAN DEFAULT TRUE,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                FOREIGN KEY (parent_id) REFERENCES categories(id) ON DELETE SET NULL
            )
        `);

        // Products table (depends on categories)
        this.db.exec(`
            CREATE TABLE IF NOT EXISTS products (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name VARCHAR(255) NOT NULL,
                description TEXT,
                price INTEGER NOT NULL CHECK (price > 0),
                inventory_count INTEGER DEFAULT 0 CHECK (inventory_count >= 0),
                category_id INTEGER,
                image_url VARCHAR(500),
                sku VARCHAR(100) UNIQUE,
                is_active BOOLEAN DEFAULT TRUE,
```

```
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (category_id) REFERENCES categories(id) ON DELETE SET NULL
        )
    `);

    // Users table (no dependencies)
    this.db.exec(`
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            email VARCHAR(255) UNIQUE NOT NULL,
            password_hash VARCHAR(255) NOT NULL,
            first_name VARCHAR(100),
            last_name VARCHAR(100),
            phone VARCHAR(20),
            is_active BOOLEAN DEFAULT TRUE,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
    `);

    // Carts table (depends on users)
    this.db.exec(`
        CREATE TABLE IF NOT EXISTS carts (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER,
            session_id VARCHAR(255),
            status VARCHAR(20) DEFAULT 'active' CHECK (status IN ('active', 'abandoned', 'converted')),
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            expires_at TIMESTAMP,
            FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
        )
```

```
        `);

        // Cart items table (depends on carts and products)
        this.db.exec(`
            CREATE TABLE IF NOT EXISTS cart_items (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                cart_id INTEGER NOT NULL,
                product_id INTEGER NOT NULL,
                quantity INTEGER NOT NULL DEFAULT 1 CHECK (quantity > 0),
                unit_price INTEGER NOT NULL CHECK (unit_price > 0),
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                FOREIGN KEY (cart_id) REFERENCES carts(id) ON DELETE CASCADE,
                FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE RESTRICT,
                UNIQUE (cart_id, product_id)
            )
        `);

        // Orders table (depends on users)
        this.db.exec(`
            CREATE TABLE IF NOT EXISTS orders (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                user_id INTEGER,
                order_number VARCHAR(50) UNIQUE NOT NULL,
                status VARCHAR(20) DEFAULT 'pending' CHECK (status IN ('pending', 'confirmed', 'shipped',
'delivered', 'cancelled')),
                subtotal INTEGER NOT NULL CHECK (subtotal > 0),
                tax_amount INTEGER DEFAULT 0 CHECK (tax_amount >= 0),
                shipping_amount INTEGER DEFAULT 0 CHECK (shipping_amount >= 0),
                total_amount INTEGER NOT NULL CHECK (total_amount > 0),
                shipping_first_name VARCHAR(100) NOT NULL,
                shipping_last_name VARCHAR(100) NOT NULL,
                shipping_address_line1 VARCHAR(255) NOT NULL,
```

```
            shipping_address_line2 VARCHAR(255),
            shipping_city VARCHAR(100) NOT NULL,
            shipping_state VARCHAR(100) NOT NULL,
            shipping_postal_code VARCHAR(20) NOT NULL,
            shipping_country VARCHAR(100) DEFAULT 'United States',
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL
        )
    `);

    // Order items table (depends on orders and products)
    this.db.exec(`
        CREATE TABLE IF NOT EXISTS order_items (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            order_id INTEGER NOT NULL,
            product_id INTEGER,
            product_name VARCHAR(255) NOT NULL,
            product_description TEXT,
            quantity INTEGER NOT NULL CHECK (quantity > 0),
            unit_price INTEGER NOT NULL CHECK (unit_price > 0),
            line_total INTEGER NOT NULL CHECK (line_total > 0),
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE,
            FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE SET NULL
        )
    `);

    this.createIndexes();
}

/**
 * Creates database indexes for common query patterns.
```

```
     * Optimizes lookups for product search, cart operations, and order history.
     */
    createIndexes() {
        const indexes = [
            'CREATE INDEX IF NOT EXISTS idx_products_category ON products(category_id)',
            'CREATE INDEX IF NOT EXISTS idx_products_active ON products(is_active)',
            'CREATE INDEX IF NOT EXISTS idx_products_sku ON products(sku)',
            'CREATE INDEX IF NOT EXISTS idx_carts_user ON carts(user_id)',
            'CREATE INDEX IF NOT EXISTS idx_carts_session ON carts(session_id)',
            'CREATE INDEX IF NOT EXISTS idx_cart_items_cart ON cart_items(cart_id)',
            'CREATE INDEX IF NOT EXISTS idx_cart_items_product ON cart_items(product_id)',
            'CREATE INDEX IF NOT EXISTS idx_orders_user ON orders(user_id)',
            'CREATE INDEX IF NOT EXISTS idx_orders_number ON orders(order_number)',
            'CREATE INDEX IF NOT EXISTS idx_order_items_order ON order_items(order_id)',
            'CREATE INDEX IF NOT EXISTS idx_users_email ON users(email)',
            'CREATE INDEX IF NOT EXISTS idx_categories_parent ON categories(parent_id)'
        ];

        indexes.forEach(indexSql => this.db.exec(indexSql));
    }

    /**
     * Seeds database with sample data for development and testing.
     * Creates sample categories, products, and a test user account.
     */
    seedInitialData() {
        // TODO 1: Insert root categories (Electronics, Clothing, Books)
        // TODO 2: Insert subcategories (Smartphones, Laptops under Electronics)
        // TODO 3: Insert sample products with realistic prices and inventory
        // TODO 4: Create a test user account with hashed password
        // Hint: Use prepared statements for data insertion to prevent SQL injection
    }
```

```
    /**
     * Returns the SQLite database connection for model classes.
     */
    getConnection() {
        return this.db;
    }

    /**
     * Closes database connection gracefully.
     */
    close() {
        if (this.db) {
            this.db.close();
        }
    }
}

module.exports = new DatabaseManager();
```

Core Model Skeleton Code

JAVASCRIPT

```javascript
// src/models/product.js

class Product {

    constructor(database) {

        this.db = database.getConnection();

    }


    /**

     * Creates a new product with category assignment and inventory.

     * Validates required fields and price precision before insertion.

     */

    create(productData) {

        // TODO 1: Validate required fields (name, price)

        // TODO 2: Check that category_id exists if provided

        // TODO 3: Generate unique SKU if not provided

        // TODO 4: Insert product record with current timestamp

        // TODO 5: Return created product with generated ID

        // Hint: Use prepared statement with placeholder parameters

    }


    /**

     * Finds products by category with pagination and sorting options.

     * Supports filtering by active status and inventory availability.

     */

    findByCategory(categoryId, options = {}) {

        // TODO 1: Build base query with category filter

        // TODO 2: Add active status filter (is_active = TRUE)

        // TODO 3: Add inventory filter if options.inStockOnly is true

        // TODO 4: Apply sorting (price, name, created_at)

        // TODO 5: Add LIMIT and OFFSET for pagination

        // Hint: Use object destructuring for options with defaults

    }


    /**
```

```
     * Updates product inventory after purchase or restocking.
     * Uses atomic operations to prevent race conditions.
     */
    updateInventory(productId, quantityChange) {
        // TODO 1: Start database transaction
        // TODO 2: Get current inventory with FOR UPDATE lock
        // TODO 3: Calculate new inventory count
        // TODO 4: Validate new count is not negative
        // TODO 5: Update inventory and commit transaction
        // Hint: Use transaction.prepare() for atomic updates
    }
}

module.exports = Product;
```

Language-Specific Database Tips

- Use `better-sqlite3` for development as it provides excellent performance and doesn't require a separate database server
- Always use prepared statements with parameter binding: `stmt.run(param1, param2)` instead of string concatenation
- Enable foreign key constraints with `PRAGMA foreign_keys = ON` to enforce referential integrity
- Use transactions for multi-table operations: `db.transaction(() => { /* multiple operations */ })()`
- Store all monetary values as integers representing cents to avoid floating-point precision issues
- Use `TIMESTAMP DEFAULT CURRENT_TIMESTAMP` for automatic timestamp management
- Consider using `UPSERT` operations (`INSERT ... ON CONFLICT`) for cart item updates

Milestone Checkpoints

After implementing the data model, verify the following functionality:

1. **Schema Creation**: Run `node -e "require('./src/models/database').initialize()"` - should create all tables without errors
2. **Foreign Key Enforcement**: Try inserting a product with invalid category_id - should fail with constraint error
3. **Constraint Validation**: Try inserting negative price or inventory - should fail with CHECK constraint error
4. **Index Performance**: Use `EXPLAIN QUERY PLAN` to verify indexes are used for category and user lookups
5. **Data Seeding**: Run seed data script - should create sample categories, products, and test user

Common Data Model Pitfalls

⚠️ **Pitfall: Using FLOAT for Monetary Values** Many developers initially use FLOAT or DECIMAL types for prices, leading to precision errors in calculations. For example, 0.1 + 0.2 = 0.30000000000000004 in floating-point arithmetic. Store prices as integers in cents (e.g., $29.99 = 2999) and convert to dollars only for display.

⚠️ **Pitfall: Missing Inventory Constraints** Forgetting to add CHECK constraints allows negative inventory counts, causing overselling situations. Always include `CHECK (inventory_count >= 0)` and use transactions for inventory updates to prevent race conditions.

⚠️ **Pitfall: Inadequate Foreign Key Actions** Using default CASCADE actions can cause unintended data loss. For example, deleting a category shouldn't delete all products - use `SET NULL` instead. Design constraint actions based on business requirements, not database defaults.

⚠️ **Pitfall: Missing Updated Timestamp Maintenance** SQLite doesn't automatically update `updated_at` fields on record changes. Implement triggers or update timestamps manually in application code to maintain accurate audit trails for data changes.

Product Catalog Component

> **Milestone(s):** Milestone 1 (Product Catalog) - establishes the foundation for displaying products with images, prices, and descriptions, including pagination, sorting, category filtering, and search functionality

The product catalog serves as the digital storefront of the e-commerce platform, presenting products to customers in an organized, searchable format. This component must handle the complex challenge of efficiently organizing thousands of products while providing fast, intuitive discovery mechanisms that help customers find exactly what they need.

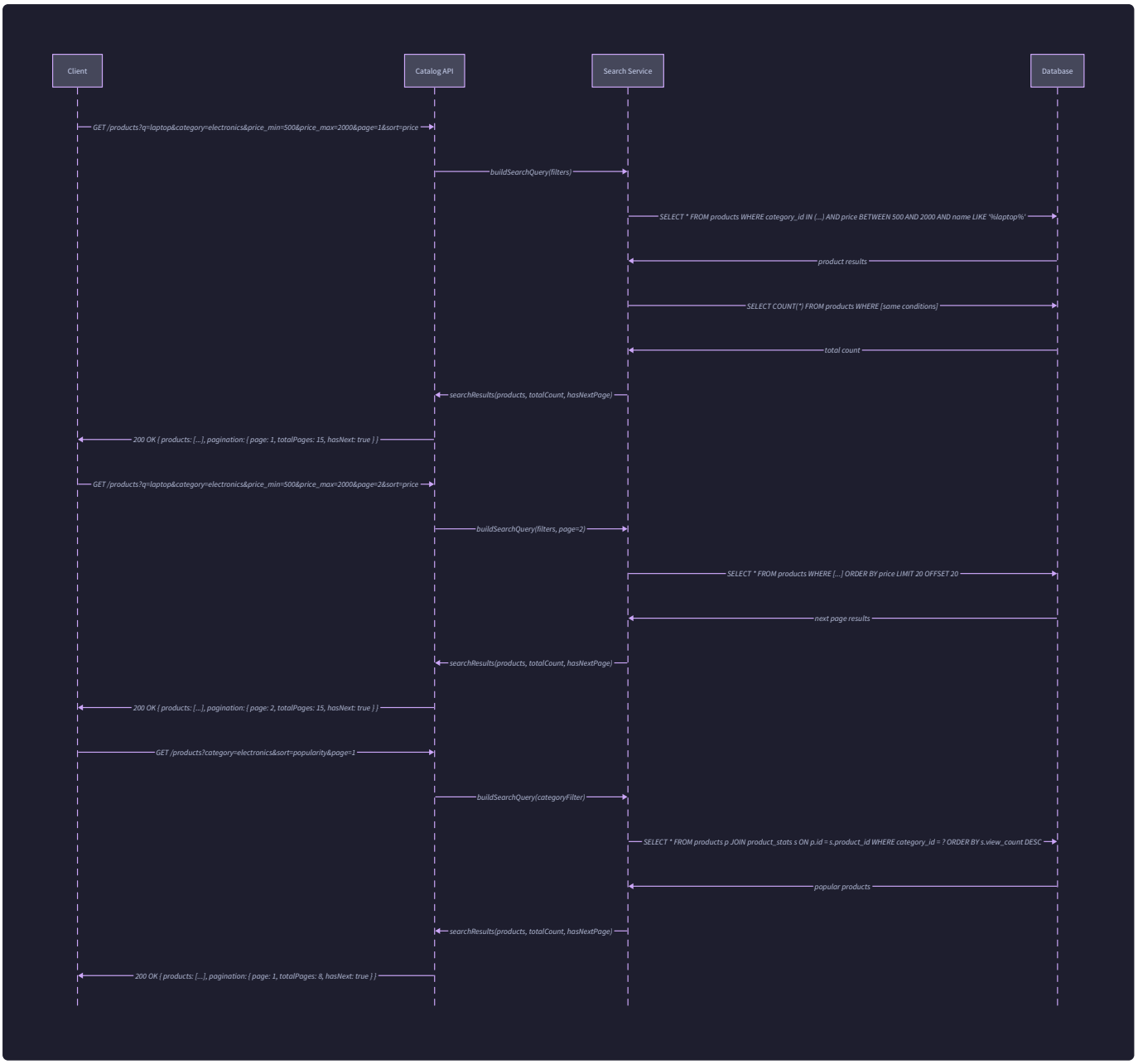
Catalog Mental Model: Library Catalog System Analogy

Think of the product catalog as a modern library catalog system, but instead of books, we're organizing products for sale. Just as a library uses the Dewey Decimal System to categorize books into hierarchical categories (000-999, with subcategories like 796.332 for American Football), our product catalog uses a **hierarchical structure** of categories and subcategories to organize merchandise.

In a physical library, patrons can browse shelves by category, use card catalogs to search by author or title, and rely on librarians to help locate specific items. Similarly, our digital catalog provides multiple discovery paths: customers can browse by category hierarchy, search by keywords, filter by attributes like price range, and sort results by relevance, price, or popularity.

The key insight is that effective product discovery requires multiple organizational schemes working together. A single product might be findable through category browsing (Electronics → Laptops → Gaming Laptops), keyword search ("gaming laptop RTX"), attribute filtering (price $1000-$2000, brand "ASUS"), and algorithmic recommendations. Like a well-organized library, the catalog must support all these access patterns efficiently.

The catalog also maintains an "inventory ledger" - tracking how many units of each product are available, similar to how libraries track which books are checked out versus available on the shelf. This **inventory consistency** becomes critical when multiple customers are shopping simultaneously, requiring careful coordination to prevent overselling.



Catalog Interface

The catalog exposes its functionality through a comprehensive API that supports all product discovery patterns. Each endpoint is designed to handle the specific needs of different user interface components while maintaining consistent data formats and error handling.

Endpoint	Method	Parameters	Returns	Description
`/api/products`	GET	`page`, `limit`, `sort`, `category`, `search`, `minPrice`, `maxPrice`	`ProductListResponse`	Returns paginated list of products with filtering and sorting options
`/api/products/:id`	GET	`id` (path parameter)	`ProductDetailResponse`	Returns detailed information for a specific product including inventory status
`/api/categories`	GET	`parentId` (optional)	`CategoryListResponse`	Returns category hierarchy, optionally filtered to children of specified parent
`/api/categories/:id/products`	GET	`id` (path parameter), pagination params	`ProductListResponse`	Returns products within a specific category branch including subcategories
`/api/search/suggestions`	GET	`query`, `limit`	`SearchSuggestionResponse`	Returns autocomplete suggestions based on partial query input
`/api/products/featured`	GET	`limit`, `category` (optional)	`ProductListResponse`	Returns featured or promoted products for homepage or category landing pages

The **product listing endpoint** (`/api/products`) serves as the workhorse of the catalog, supporting complex queries that combine multiple filters. The `sort` parameter accepts values like `price_asc`, `price_desc`, `name_asc`, `popularity`, and `newest`, allowing customers to organize results according to their shopping preferences. The `search` parameter performs full-text search across product names, descriptions, and searchable attributes.

Pagination is handled through `page` and `limit` parameters, with the response including metadata about total results, current page, and available pages. This prevents performance issues when dealing with large product catalogs and provides a smooth browsing experience.

The **category filtering** works hierarchically - requesting products from category "Electronics" will include products from all subcategories like "Laptops", "Phones", "Tablets", etc. This matches customer expectations where browsing "Electronics" should show all electronic products, not just those directly tagged with the parent category.

Price filtering uses `minPrice` and `maxPrice` parameters with values in cents to avoid floating-point precision issues. The API validates that `minPrice` ≤ `maxPrice` and returns appropriate error messages for invalid ranges.

Search and Filtering Algorithm

The product search and filtering system implements a multi-stage process that combines keyword matching, attribute filtering, and result ranking to deliver relevant products efficiently. Understanding this algorithm is crucial because it directly impacts the customer experience and system performance.

Stage 1: Query Parsing and Validation

1. Parse incoming request parameters and validate data types, ranges, and required fields
2. Sanitize the search query to prevent SQL injection attacks and normalize whitespace
3. Extract filter criteria (category, price range, attributes) and pagination parameters
4. Set default values for missing parameters (page=1, limit=20, sort=relevance)

Stage 2: Category Scope Determination

1. If a category filter is specified, retrieve the category record and validate it exists
2. Build the complete category subtree by recursively finding all child categories
3. Generate a list of category IDs to include in the product filter scope
4. Cache category hierarchies to avoid repeated tree traversals for common categories

Stage 3: Search Index Query Construction

1. If a search query is provided, tokenize it into individual keywords and phrases
2. Apply stemming to reduce words to root forms (searching, searches → search)
3. Build search score calculations that weight matches in product names higher than descriptions
4. Construct SQL LIKE patterns or full-text search queries depending on database capabilities

Stage 4: Filter Application

1. Apply category filters using the category ID list from Stage 2
2. Apply price range filters using the validated min/max price bounds in cents
3. Apply inventory filters to exclude out-of-stock products (if business rules require)
4. Combine all filters using SQL AND operations for intersection-based filtering

Stage 5: Sorting and Ranking

1. Calculate relevance scores for search results based on keyword match positions and frequency
2. Apply the requested sort order (price, name, popularity, newest, or relevance)
3. Handle secondary sort criteria (e.g., sort by price, then by name for ties)
4. Optimize sort operations using database indexes on commonly sorted columns

Stage 6: Pagination and Result Assembly

1. Calculate offset and limit values based on requested page and page size
2. Execute the final database query with all filters, sorting, and pagination applied
3. Retrieve product records and transform them into the API response format
4. Calculate pagination metadata (total results, total pages, has next/previous page)

Stage 7: Response Enhancement

1. Load associated data like category names and primary product images
2. Apply business rules like promotional pricing or member discounts
3. Add computed fields like average ratings or review counts if available
4. Format prices for display and ensure all monetary values are properly rounded

This **step-by-step process** ensures consistent results while maintaining good performance characteristics. The algorithm is designed to fail fast - validation errors are caught early, and expensive operations like full-text search are only performed when necessary.

> **Key Design Insight**: The search algorithm prioritizes precision over recall in the early stages, then expands results if needed. It's better to show 10 highly relevant products than 100 marginally relevant ones, as customers typically refine their searches rather than browse through many pages of results.

Catalog Architecture Decisions

The catalog component involves several critical architectural decisions that significantly impact performance, maintainability, and user experience. Each decision represents a trade-off between competing concerns, and understanding these trade-offs is essential for building a robust system.

> **Decision: Database Schema Design for Product Categories**
>
> - **Context**: Products need to be organized in hierarchical categories (Electronics → Laptops → Gaming Laptops), and customers expect to browse category trees while also searching across multiple categories simultaneously.
> - **Options Considered**: Adjacency List (parent_id column), Nested Sets (left/right boundaries), Path Enumeration (materialized paths), Closure Table (separate junction table)
> - **Decision**: Adjacency List with materialized path caching
> - **Rationale**: Adjacency List provides the simplest mental model and easiest mutations (adding/moving categories), while materialized path caching solves the performance problem of recursive queries without the complexity of nested sets
> - **Consequences**: Category hierarchy queries are fast for reads, category mutations require cache invalidation, and the schema remains intuitive for developers

Schema Approach	Pros	Cons	Query Performance
Adjacency List	Simple schema, easy mutations, intuitive	Requires recursive queries for trees	O(n) for deep hierarchies
Nested Sets	Excellent read performance	Complex mutations, hard to understand	O(1) for subtrees
Materialized Path	Good balance, supports path queries	Path string parsing required	O(log n) with indexes
Chosen: Adjacency + Cache	**Simple + fast reads**	**Cache invalidation complexity**	**O(1) cached, O(n) uncached**

Decision: Product Image Storage and Optimization Strategy

- **Context**: E-commerce platforms require high-quality product images in multiple sizes (thumbnails, detail views, zoom), and images significantly impact page load times and conversion rates
- **Options Considered**: Database BLOB storage, Local file storage with web server, Cloud storage (AWS S3) with CDN, Image optimization service
- **Decision**: Cloud storage with CDN and on-demand image resizing
- **Rationale**: Separates image serving from application servers, provides global CDN distribution for fast loading, enables automatic optimization for different devices, and scales independently of application infrastructure
- **Consequences**: Requires external service configuration, introduces dependency on cloud provider, but dramatically improves performance and reduces server load

Decision: Search Implementation Approach

- **Context**: Customers need to search across product names, descriptions, categories, and attributes with reasonable performance and relevance ranking
- **Options Considered**: Database LIKE queries, Full-text search extensions (PostgreSQL FTS), External search engines (Elasticsearch), Hybrid approach
- **Decision**: Database full-text search with search result caching for beginner implementation
- **Rationale**: Minimizes external dependencies while providing adequate search functionality, allows the project to focus on core e-commerce concepts rather than search infrastructure complexity
- **Consequences**: Search performance may degrade with very large catalogs, advanced search features are limited, but implementation complexity remains manageable for learning purposes

The **pagination strategy** requires careful consideration of performance implications. Deep pagination (page 1000 of results) becomes expensive with OFFSET-based queries, but most e-commerce customers rarely browse beyond the first few pages. The system uses OFFSET/LIMIT for simplicity while monitoring query performance.

Decision: Price Precision and Storage Format

- **Context**: Monetary calculations require exact precision to avoid rounding errors that can cause accounting discrepancies or customer complaints
- **Options Considered**: Floating-point numbers (DECIMAL/FLOAT), Integer cents, Currency-specific libraries, Fixed-point arithmetic
- **Decision**: Integer storage in cents with currency-aware formatting
- **Rationale**: Eliminates floating-point precision errors, simplifies arithmetic operations, works consistently across programming languages, and provides exact monetary calculations
- **Consequences**: All price calculations use integer arithmetic, display formatting converts cents to dollars, and currency conversions require explicit handling

Inventory tracking integration with the catalog requires atomic operations to prevent race conditions. When multiple customers view a product simultaneously, the displayed inventory levels must remain consistent, and the system must prevent overselling during high-traffic periods.

The catalog uses **optimistic concurrency control** for inventory updates - reading the current inventory level, performing calculations, and updating with a version check. If another transaction modifies inventory between the read and update, the operation retries with fresh data.

Concurrency Strategy	Use Case	Trade-offs	Implementation
Pessimistic Locking	High contention scenarios	Blocks readers, deadlock risk	SELECT FOR UPDATE
Optimistic Locking	**Normal e-commerce traffic**	**Retry complexity, good throughput**	**Version columns + CAS**
Queue-based Updates	Very high write volume	Eventual consistency, complexity	Background job processors

Common Catalog Pitfalls

Building an effective product catalog involves numerous subtle challenges that can significantly impact performance, user experience, and data consistency. Understanding these pitfalls helps developers avoid common mistakes that often only surface under production load or edge cases.

⚠️ Pitfall: N+1 Query Problem in Product Listings

The most common performance issue occurs when displaying product lists that include related data like categories, images, or ratings. The naive approach loads products first, then makes separate database queries for each product's related data, resulting in 1 query for products + N queries for related data.

For a page showing 20 products, this creates 21 database queries instead of 2-3 optimized queries. With hundreds of concurrent users, this quickly overwhelms the database connection pool and creates unacceptable response times.

Detection: Monitor database query counts per request. If you see query counts that scale linearly with result set size, you likely have N+1 problems. Database query logs will show repeated similar queries with different IDs.

Solution: Use explicit JOIN queries or batch loading to fetch related data in a single query. For example, instead of loading products then querying categories for each product, JOIN the products and categories tables to load everything together. Alternatively, use an ORM's "eager loading" or "include" functionality to batch related queries.

⚠️ Pitfall: Price Precision Errors and Rounding Issues

Storing prices as floating-point numbers (DECIMAL(8,2) or JavaScript Number) leads to precision errors that compound during calculations. A product priced at $19.99 might become $19.990000001 or $19.98999999 after tax calculations, leading to incorrect totals and customer complaints.

Example scenario: A customer adds 3 items at $19.99 each. The subtotal should be $59.97, but floating-point arithmetic might calculate $59.970000003 or $59.969999997. When displayed as $59.97, the internal calculation discrepancy causes audit failures.

Solution: Store all monetary values as integers in the smallest currency unit (cents for USD). A $19.99 product is stored as 1999 cents. All arithmetic uses integer operations, eliminating precision errors. Format prices for display by dividing by 100 and adding currency symbols.

⚠️ Pitfall: Inefficient Image Loading and Storage

Storing product images as database BLOBs or serving full-resolution images for thumbnails creates severe performance bottlenecks. Database servers aren't optimized for binary data serving, and large images consume excessive bandwidth and slow page loading.

A common mistake is storing a single high-resolution image (2MB) and resizing it client-side for thumbnails. This forces every product listing to download 2MB per product thumbnail, making category pages unusably slow.

Solution: Generate multiple image sizes during upload (thumbnail 150x150, medium 400x400, large 800x800, zoom 1200x1200) and store them in cloud storage with CDN distribution. Reference images by URL in the database rather than storing binary data. Implement lazy loading for images below the fold.

⚠️ Pitfall: Inadequate Search Performance with Large Catalogs

Using SQL LIKE queries for product search works adequately for small catalogs but becomes unusably slow as the product count grows. A query like `WHERE name LIKE '%laptop%' OR description LIKE '%laptop%'` cannot use database indexes effectively and requires full table scans.

Detection: Search queries that take multiple seconds to complete, especially as catalog size grows. Database query plans showing full table scans rather than index usage.

Solution: Implement proper full-text search using database-specific features (PostgreSQL's `to_tsvector`, MySQL's `FULLTEXT` indexes) or dedicated search engines. Create composite indexes on commonly searched combinations of fields. Cache popular search results to avoid repeated expensive queries.

⚠️ Pitfall: Category Hierarchy Query Performance

Retrieving all products in a category tree (Electronics and all subcategories) using recursive queries or multiple database round-trips becomes expensive with deep hierarchies. The naive approach queries each category level separately, creating O(depth) database queries.

Solution: Denormalize category paths by storing the complete hierarchy path with each product (`/electronics/computers/laptops/`) or maintain a separate category closure table that pre-computes all ancestor-descendant relationships. This trades storage space for query performance.

⚠️ Pitfall: Inconsistent Inventory Display During Concurrent Access

When multiple customers view the same product simultaneously, they might see different inventory levels due to race conditions between inventory reads and updates. Customer A sees "5 in stock," Customer B sees "3 in stock" a few seconds later, causing confusion and potential overselling.

Detection: Customer service complaints about inventory discrepancies, audit logs showing inventory levels that don't match expected values.

Solution: Implement proper transaction isolation for inventory reads and updates. Use database transactions with appropriate isolation levels, and consider implementing inventory reservation systems where adding items to cart temporarily reserves inventory for a short period.

⚠️ Pitfall: Poor Pagination Performance with Large Datasets

Using OFFSET-based pagination (`LIMIT 20 OFFSET 10000`) becomes extremely slow for deep pages because the database must count and skip thousands of records. Page 500 of search results might take 10+ seconds to load.

Solution: Implement cursor-based pagination using stable sort keys (like product ID). Instead of `OFFSET 10000`, use `WHERE id > last_seen_id ORDER BY id LIMIT 20`. This maintains constant-time performance regardless of page depth.

Implementation Guidance

The catalog component serves as the foundation of the e-commerce platform, requiring careful attention to data modeling, API design, and performance optimization. The following implementation guidance provides concrete direction for building a robust product catalog system.

Technology Recommendations:

Component	Simple Option	Advanced Option
Database	SQLite with FTS5	PostgreSQL with full-text search
Image Storage	Local file storage	AWS S3 with CloudFront CDN
Search Engine	Database full-text search	Elasticsearch with relevance tuning
Caching	In-memory JavaScript objects	Redis with TTL-based invalidation
Image Processing	Sharp.js for Node.js	ImageMagick with multiple size generation

Recommended File Structure:

```
project-root/
├── src/
│   ├── models/
│   │   ├── Product.js          ← Product entity and database operations
│   │   ├── Category.js         ← Category hierarchy management
│   │   └── Database.js         ← Database connection and initialization
│   ├── controllers/
│   │   ├── ProductController.js    ← Product catalog API endpoints
│   │   └── CategoryController.js   ← Category browsing endpoints
│   ├── services/
│   │   ├── SearchService.js        ← Search and filtering logic
│   │   └── ImageService.js         ← Image upload and optimization
│   ├── middleware/
│   │   ├── validation.js           ← Request parameter validation
│   │   └── pagination.js           ← Pagination helper functions
│   ├── routes/
│   │   └── catalog.js              ← Catalog route definitions
│   └── app.js                      ← Main application setup
├── uploads/
│   └── products/                   ← Product image storage
├── database/
│   ├── migrations/                 ← Database schema changes
│   └── seeds/                      ← Sample product data
└── tests/
    └── catalog/                    ← Catalog component tests
```

Core Database Schema:

JAVASCRIPT

```javascript
// Database.js - Complete database setup with proper constraints

const sqlite3 = require('sqlite3').verbose();

const path = require('path');


class Database {

    constructor() {

        this.db = null;

    }


    initialize() {

        const dbPath = process.env.NODE_ENV === 'test' ? ':memory:' : path.join(__dirname,
'../database/ecommerce.db');

        this.db = new sqlite3.Database(dbPath);



        // Enable foreign key constraints

        this.db.run('PRAGMA foreign_keys = ON');



        return this.createTables();

    }


    createTables() {

        return new Promise((resolve, reject) => {

            const schema = `

                CREATE TABLE IF NOT EXISTS categories (

                    id INTEGER PRIMARY KEY AUTOINCREMENT,

                    name TEXT NOT NULL,

                    parent_id INTEGER,

                    path TEXT NOT NULL,

                    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,

                    FOREIGN KEY (parent_id) REFERENCES categories(id),

                    UNIQUE(name, parent_id)

                );


                CREATE TABLE IF NOT EXISTS products (
```

```
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    description TEXT,
    price INTEGER NOT NULL,
    inventory_quantity INTEGER NOT NULL DEFAULT 0,
    category_id INTEGER,
    image_url TEXT,
    is_active BOOLEAN DEFAULT true,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (category_id) REFERENCES categories(id),
    CHECK (price >= 0),
    CHECK (inventory_quantity >= 0)
);

CREATE INDEX IF NOT EXISTS idx_products_category ON products(category_id);
CREATE INDEX IF NOT EXISTS idx_products_price ON products(price);
CREATE INDEX IF NOT EXISTS idx_products_active ON products(is_active);
CREATE INDEX IF NOT EXISTS idx_categories_parent ON categories(parent_id);
CREATE INDEX IF NOT EXISTS idx_categories_path ON categories(path);

-- Full-text search virtual table
CREATE VIRTUAL TABLE IF NOT EXISTS products_fts USING fts5(
    name, description, content=products, content_rowid=id
);

-- Triggers to maintain FTS index
CREATE TRIGGER IF NOT EXISTS products_fts_insert AFTER INSERT ON products
BEGIN
    INSERT INTO products_fts(rowid, name, description)
    VALUES (new.id, new.name, new.description);
END;

CREATE TRIGGER IF NOT EXISTS products_fts_update AFTER UPDATE ON products
```

```
                BEGIN
                    UPDATE products_fts SET name = new.name, description = new.description
                    WHERE rowid = new.id;
                END;

                CREATE TRIGGER IF NOT EXISTS products_fts_delete AFTER DELETE ON products
                BEGIN
                    DELETE FROM products_fts WHERE rowid = old.id;
                END;
            `;

            this.db.exec(schema, (err) => {
                if (err) reject(err);
                else resolve();
            });
        });
    }

    seedInitialData() {
        const categories = [
            { name: 'Electronics', parent_id: null, path: '/electronics' },
            { name: 'Laptops', parent_id: 1, path: '/electronics/laptops' },
            { name: 'Phones', parent_id: 1, path: '/electronics/phones' },
            { name: 'Clothing', parent_id: null, path: '/clothing' },
            { name: 'Shirts', parent_id: 4, path: '/clothing/shirts' }
        ];

        const products = [
            { name: 'Gaming Laptop', description: 'High-performance laptop for gaming', price: 129999,
inventory_quantity: 10, category_id: 2 },
            { name: 'Business Laptop', description: 'Professional laptop for work', price: 89999,
inventory_quantity: 15, category_id: 2 },
            { name: 'Smartphone', description: 'Latest Android smartphone', price: 79999, inventory_quantity:
25, category_id: 3 },
            { name: 'Cotton T-Shirt', description: 'Comfortable cotton t-shirt', price: 1999,
inventory_quantity: 50, category_id: 5 }
```

```
        ];

        // Insert sample data (implementation details omitted for brevity)
        // This would use prepared statements to safely insert the sample data
    }

    getConnection() {
        return this.db;
    }

    close() {
        return new Promise((resolve) => {
            this.db.close(resolve);
        });
    }
}

module.exports = Database;
```

Product Model with Search Capabilities:

JAVASCRIPT

```javascript
// models/Product.js - Core product operations skeleton

class Product {

    constructor(database) {

        this.db = database.getConnection();

    }


    // Find products with comprehensive filtering and search

    async findWithFilters(options = {}) {

        // TODO 1: Extract and validate filter parameters (search, category, price range, pagination)

        // TODO 2: Build base query with JOINs for category information

        // TODO 3: Add search conditions using FTS if search query provided

        // TODO 4: Add category filter including subcategories if category specified

        // TODO 5: Add price range filtering if minPrice or maxPrice provided

        // TODO 6: Apply sorting based on sort parameter (price_asc, price_desc, name_asc, relevance)

        // TODO 7: Add pagination with LIMIT and OFFSET

        // TODO 8: Execute query and return formatted results with metadata



        // Hint: Use parameterized queries to prevent SQL injection

        // Hint: Calculate total count for pagination metadata

        // Hint: Format prices from cents to dollars in response

    }


    // Get single product with full details

    async findById(productId) {

        // TODO 1: Validate productId is a positive integer

        // TODO 2: Query product with JOIN to get category information

        // TODO 3: Check if product exists and is active

        // TODO 4: Format response with proper price formatting

        // TODO 5: Include inventory availability status



        // Hint: Return null if product not found rather than throwing error

        // Hint: Include category path for breadcrumb navigation

    }
```

```
    // Create new product with validation
    async create(productData) {
        // TODO 1: Validate required fields (name, price, category_id)
        // TODO 2: Validate price is positive integer in cents
        // TODO 3: Validate category_id exists in categories table
        // TODO 4: Insert product record with current timestamp
        // TODO 5: Return created product with generated ID

        // Hint: Use prepared statements for safe parameter binding
        // Hint: Wrap in transaction if additional operations needed
    }

    // Update inventory atomically to prevent race conditions
    async updateInventory(productId, quantityChange) {
        // TODO 1: Start database transaction for atomic operation
        // TODO 2: Lock product record for update (SELECT FOR UPDATE)
        // TODO 3: Validate current inventory + change >= 0
        // TODO 4: Update inventory_quantity with the change amount
        // TODO 5: Commit transaction and return new inventory level
        // TODO 6: Rollback transaction if any step fails

        // Hint: Use absolute quantity changes (e.g., -1 for purchase, +5 for restock)
        // Hint: Return error if insufficient inventory for negative changes
    }
}

module.exports = Product;
```

Search Service Implementation:

JAVASCRIPT

```javascript
// services/SearchService.js - Search and filtering logic skeleton

class SearchService {

    constructor(database) {

        this.db = database.getConnection();

    }


    // Execute product search with ranking and filtering

    async searchProducts(searchQuery, filters = {}) {

        // TODO 1: Sanitize and validate search query to prevent injection

        // TODO 2: Tokenize search query into individual keywords

        // TODO 3: Build FTS query using products_fts virtual table

        // TODO 4: Calculate relevance scores based on match positions

        // TODO 5: Apply additional filters (category, price, inventory)

        // TODO 6: Sort results by relevance score then secondary criteria

        // TODO 7: Apply pagination and return formatted results



        // Hint: Use FTS5 match expressions like 'laptop AND gaming'

        // Hint: Boost matches in product names over description matches

        // Hint: Cache popular search results to improve performance

    }


    // Get search suggestions for autocomplete

    async getSuggestions(partialQuery, limit = 10) {

        // TODO 1: Validate and sanitize partial query input

        // TODO 2: Query products_fts for prefix matches

        // TODO 3: Extract unique product name prefixes

        // TODO 4: Rank suggestions by product popularity/sales

        // TODO 5: Return limited list of completion suggestions



        // Hint: Use FTS prefix matching with '*' operator

        // Hint: Deduplicate suggestions from multiple product matches

    }

}
```

```
module.exports = SearchService;
```

Category Management:

JAVASCRIPT

```
// models/Category.js - Category hierarchy operations

class Category {

    constructor(database) {

        this.db = database.getConnection();

    }


    // Get category tree structure

    async getHierarchy(parentId = null) {

        // TODO 1: Query categories filtered by parent_id

        // TODO 2: Recursively build tree structure with children

        // TODO 3: Include product counts for each category

        // TODO 4: Sort categories by name within each level

        // TODO 5: Return nested category objects with metadata



        // Hint: Use recursive CTE or multiple queries for tree building

        // Hint: Cache category trees since they change infrequently

    }


    // Get all subcategory IDs for filtering

    async getSubcategoryIds(categoryId) {

        // TODO 1: Validate categoryId exists in database

        // TODO 2: Query all categories where path starts with parent path

        // TODO 3: Extract category IDs from path-based results

        // TODO 4: Include the parent category ID in results

        // TODO 5: Return array of category IDs for filtering



        // Hint: Use path column with LIKE '/electronics%' pattern

        // Hint: This enables efficient category tree filtering

    }

}


module.exports = Category;
```

Milestone Checkpoint:

After implementing the product catalog component, verify the following functionality:

1. **Database Setup**: Run `node -e "const db = require('./src/models/Database'); new db().initialize().then(() => console('Database initialized'))"` - should create tables without errors
2. **Product Listing**: Start the server and visit `http://localhost:3000/api/products` - should return JSON array of products with pagination metadata
3. **Category Filtering**: Test `http://localhost:3000/api/products?category=1` - should return only products from the specified category and its subcategories
4. **Search Functionality**: Test `http://localhost:3000/api/products?search=laptop` - should return products matching the search term with relevance ranking
5. **Price Filtering**: Test `http://localhost:3000/api/products?minPrice=5000&maxPrice=10000` - should return products within the price range (prices in cents)

Expected Behavior Indicators:

- Product prices are stored and returned as integers (cents) to avoid precision errors
- Category filtering includes products from subcategories, not just direct matches
- Search results show most relevant products first, with name matches ranked higher than description matches
- Pagination metadata includes `totalCount`, `currentPage`, `totalPages`, and `hasNext`/`hasPrevious` flags
- All database queries use parameterized statements to prevent SQL injection

Common Issues and Debugging:

- If products aren't appearing in search results, check that the FTS triggers are properly maintaining the `products_fts` table
- If category filtering isn't working hierarchically, verify the category `path` column is being populated correctly during category creation
- If pagination is slow, ensure indexes are created on commonly filtered columns like `category_id`, `price`, and `is_active`

Shopping Cart Component

> **Milestone(s):** Milestone 2 (Shopping Cart) - implements add to cart, update quantities, and remove items functionality with cart persistence across browser sessions and proper inventory validation

Cart Mental Model: Physical Shopping Cart Analogy for State Management

Think of a shopping cart in a physical grocery store. When you enter the store, you grab a cart that becomes uniquely yours for that shopping session. As you walk through the aisles, you add items to your cart, sometimes changing your mind and removing items, or deciding you need more of something and adding additional quantities. The cart maintains its contents as you move around the store, and you can always look inside to see what you've collected and the running total of your intended purchases.

The digital shopping cart mirrors this physical experience with some important enhancements. Unlike a physical cart that you abandon when you leave the store, a digital cart can remember its contents even if you close your browser and return hours later. The cart also provides real-time inventory validation - if someone else purchases the last item you have in your cart, the system can alert you immediately rather than waiting until checkout.

The key insight for shopping cart state management is that the cart represents **intention to purchase** rather than **commitment to purchase**. Items in the cart are temporarily reserved in the user's session but remain available to other customers until the checkout process begins. This distinction drives many of the design decisions around inventory validation, price updates, and session management.

A shopping cart exists in several distinct states throughout its lifecycle. When first created, the cart is empty and tied either to an authenticated user account or an anonymous browser session. As items are added, the cart becomes active and begins tracking quantities, calculating subtotals, and maintaining references to current product prices. If left inactive for extended periods, the cart may transition to an abandoned state where cleanup processes can eventually remove it. Finally, during successful checkout, the cart converts to an order and is cleared for the next shopping session.

Cart Interface: Operations for Adding, Updating, and Removing Items

The shopping cart component exposes a well-defined interface for manipulating cart contents and querying cart state. Each operation maintains data consistency while providing immediate feedback to the user about the success or failure of their requested action.

Method Name	Parameters	Returns	Description
`addItem`	`sessionId`, `productId`, `quantity`	`CartItem`	Adds specified quantity of product to cart, creating new item or updating existing quantity
`removeItem`	`sessionId`, `cartItemId`	`boolean`	Removes specific cart item completely, recalculating cart totals
`updateQuantity`	`sessionId`, `cartItemId`, `newQuantity`	`CartItem`	Updates cart item to specified quantity, validating against available inventory
`getCartContents`	`sessionId`	`CartSummary`	Retrieves complete cart contents with current prices and availability status
`clearCart`	`sessionId`	`boolean`	Removes all items from cart, typically called after successful checkout
`validateInventory`	`sessionId`	`ValidationReport`	Checks all cart items against current inventory levels and price changes
`calculateTotals`	`sessionId`	`CartTotals`	Computes subtotal, taxes, shipping estimates, and final total
`transferCart`	`anonymousSessionId`, `userId`	`boolean`	Moves cart contents from anonymous session to authenticated user account

The `addItem` operation demonstrates the complexity hidden behind a simple interface. When a user clicks "Add to Cart", the system must first validate that the requested quantity is available in inventory. If the product already exists in the cart, the operation becomes an update rather than an insert, requiring the system to check whether the combined quantity (existing + new) exceeds available stock. The operation must also capture the current product price at the time of addition, creating a snapshot that protects the user from price increases during their shopping session while allowing the merchant to update displayed prices for new customers.

The `updateQuantity` operation handles both increases and decreases in item quantities. When increasing quantities, the system validates inventory availability just as in `addItem`. When decreasing quantities, the system must decide whether to allow the reduction immediately or require additional confirmation for significant changes. Setting quantity to zero is functionally equivalent to `removeItem` but provides better user experience by maintaining the item's position in the cart temporarily in case the user changes their mind.

Cart validation represents a critical background operation that maintains cart integrity without disrupting the user experience. The `validateInventory` method runs periodically and before checkout to identify items that are no longer available, have reduced

availability, or have changed prices significantly. Rather than automatically removing items or updating prices, the validation process flags items for user attention and provides clear options for resolution.

Cart Item State	Validation Action	User Notification	Available Options
`VALID`	No action required	None	Continue shopping
`QUANTITY_REDUCED`	Flag for attention	"Only X remaining"	Update quantity or remove
`PRICE_INCREASED`	Flag for attention	"Price changed from $X to $Y"	Accept new price or remove
`PRICE_DECREASED`	Update automatically	"Price reduced to $X"	Continue with better price
`OUT_OF_STOCK`	Mark unavailable	"No longer available"	Remove from cart
`DISCONTINUED`	Mark unavailable	"Product discontinued"	Remove or suggest alternatives

Cart State Management: Session Persistence and Synchronization Strategies

Shopping cart state management requires balancing immediate responsiveness with persistent storage, handling both authenticated and anonymous users, and maintaining consistency across multiple browser tabs or devices. The cart component must persist user intentions across browser sessions while accommodating the temporary nature of shopping decisions.

For anonymous users, the cart initially exists only in browser session storage, providing immediate responsiveness and avoiding database overhead for casual browsers. JavaScript session storage persists cart contents across page refreshes but clears when the browser tab closes, reflecting the temporary nature of anonymous shopping sessions. However, as the cart accumulates items and the user invests time in product selection, the system promotes the cart to server-side storage tied to a generated session identifier.

> **Decision: Hybrid Client-Server Cart Storage**
>
> - **Context**: Need to balance performance for casual browsers with persistence for committed shoppers, while supporting both anonymous and authenticated users across devices
> - **Options Considered**: Pure client-side storage, pure server-side storage, hybrid approach with promotion threshold
> - **Decision**: Start with client-side storage, promote to server after first item addition or 5-minute session duration
> - **Rationale**: Eliminates database load for bounce visitors while ensuring committed shoppers don't lose cart contents, supports seamless transition from anonymous to authenticated
> - **Consequences**: Requires synchronization logic between client and server, temporary inconsistency during promotion process, additional complexity in session management

Storage Strategy	Trigger Condition	Storage Location	Persistence Duration	Sync Frequency
`CLIENT_ONLY`	Empty cart, new session	Browser sessionStorage	Until tab close	N/A
`CLIENT_PROMOTED`	First item added	sessionStorage + Database	30 days inactive	On every mutation
`SERVER_AUTHORITATIVE`	User authentication	Database primary	Account lifetime	Real-time
`MULTI_DEVICE`	Multiple active sessions	Database + Redis cache	Session duration	Immediate

Authenticated users receive enhanced cart persistence with cross-device synchronization. When a user logs in from a new device, the system merges any existing anonymous cart with their persistent account cart, handling conflicts through user-controlled resolution. The merge process prioritizes user choice over automatic resolution, presenting clear options when the same product exists in both carts with different quantities or when the combined cart exceeds inventory limits.

Cart synchronization between multiple browser tabs or devices requires careful coordination to prevent lost updates and provide consistent user experience. The system implements optimistic locking with conflict detection, allowing users to modify cart contents freely while detecting when simultaneous changes occur across sessions. When conflicts arise, the system preserves all user actions and provides clear resolution options rather than silently discarding changes.

> The critical insight for cart synchronization is that users expect their actions to be preserved, even when conflicts occur. Automatic conflict resolution that discards user actions creates frustration and lost sales, while clear conflict presentation with user-controlled resolution maintains trust and shopping momentum.

Conflict Scenario	Detection Method	Resolution Strategy	User Experience
`QUANTITY_CONFLICT`	Version comparison	Present both quantities	"You have 2 in cart on mobile, 3 on desktop. Choose quantity:"
`ITEM_ADDED_ELSEWHERE`	Cart hash mismatch	Merge automatically	"Added 1 item from your mobile session"
`ITEM_REMOVED_ELSEWHERE`	Missing item reference	Confirm removal	"Item removed on another device. Restore?"
`INVENTORY_EXHAUSTED`	Validation failure	Reduce to available	"Reduced quantity to 2 (maximum available)"

Cart Architecture Decisions: Session Storage vs Database Persistence Trade-offs

The shopping cart component must balance several competing requirements: performance for high-traffic browsing, persistence for committed shoppers, consistency across user sessions, and scalability for concurrent operations. These requirements drive architectural decisions about storage mechanisms, caching strategies, and synchronization approaches.

> **Decision: Session-Based Cart Identification**
>
> - **Context**: Need to track cart contents for both anonymous and authenticated users, support seamless transition between states, and prevent cart conflicts
> - **Options Considered**: Cookie-based tracking, JWT tokens with embedded cart data, server-generated session IDs with database storage
> - **Decision**: Generate cryptographically secure session IDs stored in HTTP-only cookies, with server-side cart storage
> - **Rationale**: Provides security against XSS attacks, supports large carts without cookie size limits, enables server-side inventory validation and pricing updates
> - **Consequences**: Requires server-side session storage, adds database dependency for cart operations, enables better analytics and cart abandonment tracking

The session management approach directly impacts cart persistence and user experience. HTTP-only cookies prevent client-side JavaScript access, protecting against XSS attacks that could manipulate cart contents or steal session identifiers. The session identifier serves as a secure reference to server-side cart storage rather than embedding cart data directly in client-accessible storage.

Architecture Option	Pros	Cons	Chosen?
Browser localStorage	Fast access, offline support, no server load	Lost on device change, vulnerable to XSS, size limits	No
Session cookies with cart data	Simple implementation, stateless server	Size limits, security exposure, no server validation	No
Session ID + database storage	Secure, scalable, cross-device sync	Database dependency, network latency	**Yes**
Redis session store	Fast access, automatic expiry, scalability	Additional infrastructure, cache invalidation complexity	Future enhancement

Cart data persistence requires careful consideration of storage mechanisms and access patterns. The `Cart` and `CartItem` entities reside in the primary application database to maintain referential integrity with `Product` and `User` entities, enabling complex queries for inventory validation and order conversion. However, cart access patterns differ significantly from other e-commerce entities, with frequent reads and updates during active shopping sessions followed by long periods of inactivity.

Decision: Database-First Cart Storage with Caching Layer

- **Context**: Cart operations require real-time inventory validation and pricing updates, but also need sub-second response times for good user experience
- **Options Considered**: Database-only storage, cache-only with periodic persistence, write-through caching with database backup
- **Decision**: Database as authoritative source with Redis cache for active sessions
- **Rationale**: Ensures data consistency for inventory checks while providing fast access for active carts, supports automatic session cleanup through cache expiry
- **Consequences**: Requires cache synchronization logic, adds Redis dependency, increases operational complexity but significantly improves performance

The caching layer addresses the performance requirements of active shopping sessions while maintaining the data integrity guarantees needed for inventory management and order processing. Active cart sessions benefit from memory-speed access to cart contents, while inactive carts naturally age out of cache without requiring explicit cleanup processes.

Cart State	Primary Storage	Cache Strategy	Access Pattern	Cleanup Method
`ACTIVE`	Database + Redis	Write-through	High frequency R/W	TTL refresh on access
`INACTIVE`	Database only	Cache miss acceptable	Low frequency R	Periodic database cleanup
`CONVERTING`	Database transaction	No caching	Single atomic operation	Immediate on order creation
`ABANDONED`	Database historical	No caching	Analytics only	Archive after 90 days

Common Cart Pitfalls: Price Staleness, Inventory Validation, and Session Expiry

Shopping cart implementations frequently encounter specific categories of bugs and design flaws that can impact user experience, data consistency, and business operations. Understanding these pitfalls helps developers implement robust cart systems that gracefully handle edge cases and maintain user trust.

⚠️ Pitfall: Price Staleness Between Add and Checkout

Many cart implementations capture product prices when items are added but fail to handle price changes that occur during the shopping session. This creates several problematic scenarios: users may see outdated prices throughout their shopping experience, or worse, they may be charged different amounts than displayed in their cart. The issue becomes particularly acute during sales events or inventory clearances when prices change frequently.

The underlying problem stems from treating cart prices as immutable snapshots rather than references to current pricing. When a `CartItem` stores a `price_at_add_time` field, the system must decide whether to honor that historical price (protecting the user from increases) or update to current pricing (ensuring accurate business transactions). Neither approach handles all scenarios gracefully without additional complexity.

Detection: Compare cart item prices with current product prices during cart display and validation operations. Log significant price discrepancies for business analysis.

Resolution: Implement a price change notification system that alerts users to pricing updates while preserving their option to proceed at original prices for a limited time. Store both `original_price` and `current_price` in cart items, allowing users to make informed decisions about price changes.

Price Change Scenario	System Behavior	User Notification	Business Impact
Minor increase (< 5%)	Update silently, log change	None	Minimal customer impact
Significant increase (≥ 5%)	Flag for user attention	"Price increased from $X to $Y"	User choice preserves trust
Price decrease	Update automatically	"Price reduced - you save $X"	Positive user experience
Sale expiry	Honor sale price for 24 hours	"Sale price expires in X hours"	Limited business exposure

⚠️ Pitfall: Race Conditions in Concurrent Inventory Updates

Cart systems often fail to handle concurrent inventory updates correctly, leading to overselling scenarios where multiple users can add the same limited-stock item to their carts simultaneously. This occurs when cart operations check inventory availability and add items in separate, non-atomic operations, creating a window where inventory counts become inconsistent.

The problem manifests most severely during high-traffic sales events or when popular items reach low stock levels. Multiple users checking inventory simultaneously may all see availability, but their concurrent cart additions exceed actual stock levels. The issue often goes undetected until checkout, creating poor user experience when customers must remove items they believed were secured in their carts.

Detection: Monitor for negative inventory values, track the ratio of cart additions to successful checkouts for limited-stock items, and implement inventory audit processes that compare expected versus actual stock levels.

Resolution: Use database transactions with appropriate isolation levels for all cart operations that affect inventory. Implement optimistic locking on product inventory updates, and provide clear user feedback when requested quantities exceed current availability.

⚠️ Pitfall: Session Expiry Losing Cart Contents

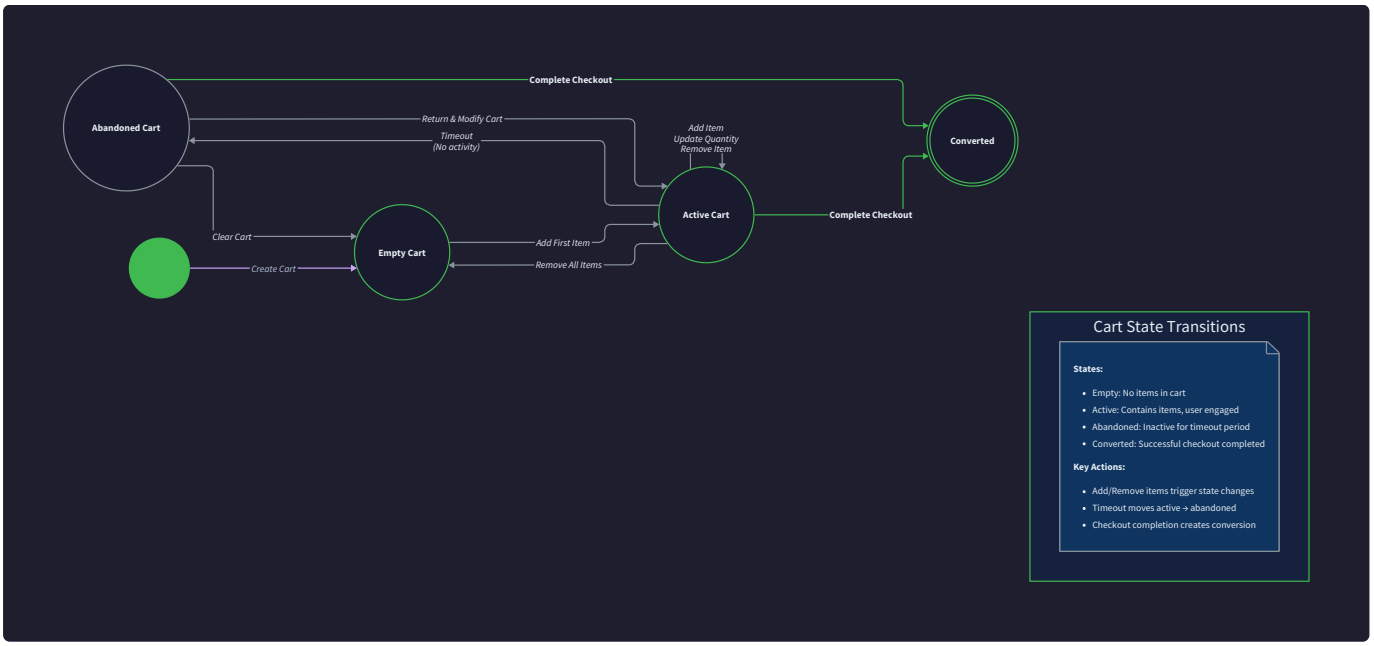
Aggressive session expiry policies can frustrate users by clearing their carefully curated cart contents, particularly for high-consideration purchases that require extended research and comparison. Many implementations use short session timeouts for security without considering the impact on shopping behavior, or fail to provide adequate warning before session expiry.

The issue is compounded by inconsistent session management across different parts of the application. Authentication sessions, cart sessions, and general browsing sessions may have different expiry policies, leading to scenarios where users remain logged in but lose their cart contents, or where cart items persist after user logout.

Detection: Track cart abandonment rates correlated with session expiry events, monitor user complaints about lost cart contents, and analyze the time distribution of successful purchases versus session timeouts.

Resolution: Implement tiered session expiry with grace periods for cart preservation. Provide clear warnings before cart expiry, allow users to extend their cart session without full re-authentication, and consider preserving cart contents for registered users across longer time periods.

Session Type	Expiry Period	Grace Period	Recovery Option
Anonymous browsing	30 minutes	10 minutes	Browser storage backup
Anonymous with cart	24 hours	4 hours	Email cart recovery link
Authenticated user	7 days	24 hours	Persistent cart storage
High-value cart (>$100)	30 days	7 days	Email reminders



Implementation Guidance

This subsection provides concrete implementation details for building a robust shopping cart component using Node.js and Express, with session management and database persistence.

Technology Recommendations

Component	Simple Option	Advanced Option
Session Management	express-session with MemoryStore	express-session with Redis store
Database Operations	Direct SQLite with better-sqlite3	Connection pooling with node-postgres
Cart Validation	Synchronous inventory checks	Background validation with job queues
Price Updates	Real-time database queries	Cached pricing with change notifications
Conflict Resolution	Last-write-wins with logging	Operational transform with merge strategies

Recommended File Structure

The cart component integrates into the existing e-commerce application structure while maintaining clear separation of concerns:

```
src/
├── components/
│   └── cart/
│       ├── CartService.js           ← Core cart business logic
│       ├── CartController.js        ← HTTP request handlers
│       ├── CartValidation.js        ← Input validation and sanitization
│       ├── SessionManager.js        ← Session lifecycle management
│       └── __tests__/
│           ├── CartService.test.js
│           └── CartController.test.js
├── middleware/
│   ├── sessionMiddleware.js         ← Express session configuration
│   └── cartMiddleware.js            ← Cart-specific request processing
├── models/
│   ├── Cart.js                      ← Cart entity model
│   └── CartItem.js                  ← CartItem entity model
└── routes/
    └── cartRoutes.js                ← Cart API endpoint definitions
```

Session Management Infrastructure

Complete Session Middleware (src/middleware/sessionMiddleware.js):

JAVASCRIPT

```javascript
const session = require('express-session');
const crypto = require('crypto');

const SESSION_SECRET = process.env.SESSION_SECRET || crypto.randomBytes(64).toString('hex');
const NODE_ENV = process.env.NODE_ENV || 'development';

const sessionConfig = {
  secret: SESSION_SECRET,
  name: 'ecommerce_session',
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: NODE_ENV === 'production',
    httpOnly: true,
    maxAge: 24 * 60 * 60 * 1000, // 24 hours
    sameSite: 'strict'
  }
};

const sessionMiddleware = session(sessionConfig);

// Cart-specific session initialization
const cartSessionMiddleware = (req, res, next) => {
  // Ensure session has cart structure
  if (!req.session.cart) {
    req.session.cart = {
      id: null,
      items: [],
      createdAt: new Date().toISOString(),
      lastModified: new Date().toISOString()
    };
  }

  // Generate cart session ID if not exists
```

```
  if (!req.session.cartId) {
    req.session.cartId = crypto.randomBytes(16).toString('hex');
  }

  next();
};

module.exports = {
  sessionMiddleware,
  cartSessionMiddleware
};
```

Database Schema for Cart Persistence (src/database/cartSchema.sql):

```sql
-- Cart entity representing user shopping session
CREATE TABLE IF NOT EXISTS carts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    session_id TEXT UNIQUE NOT NULL,
    user_id INTEGER NULL,
    status TEXT NOT NULL DEFAULT 'active',
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    expires_at DATETIME NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL,
    CHECK (status IN ('active', 'abandoned', 'converted', 'expired'))
);

-- Cart items linking products to carts with quantities
CREATE TABLE IF NOT EXISTS cart_items (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    cart_id INTEGER NOT NULL,
    product_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    price_snapshot INTEGER NOT NULL,
    added_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (cart_id) REFERENCES carts(id) ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE CASCADE,
    UNIQUE (cart_id, product_id)
);

-- Indexes for efficient cart operations
CREATE INDEX IF NOT EXISTS idx_carts_session_id ON carts(session_id);
CREATE INDEX IF NOT EXISTS idx_carts_user_id ON carts(user_id);
CREATE INDEX IF NOT EXISTS idx_carts_expires_at ON carts(expires_at);
CREATE INDEX IF NOT EXISTS idx_cart_items_cart_id ON cart_items(cart_id);
```

SQL

Core Cart Service Implementation

Cart Service Skeleton (src/components/cart/CartService.js):

JAVASCRIPT

```javascript
const Database = require('../../database/Database');

class CartService {
  constructor(database) {
    this.db = database;
  }

  // Add item to cart with inventory validation
  async addItem(sessionId, productId, quantity) {
    // TODO 1: Start database transaction for atomic operation
    // TODO 2: Retrieve or create cart for session ID with expiry extension
    // TODO 3: Validate product exists and has sufficient inventory
    // TODO 4: Check if product already exists in cart (update vs insert)
    // TODO 5: If existing, validate combined quantity against inventory
    // TODO 6: Capture current product price as price_snapshot
    // TODO 7: Insert or update cart_item record with new quantity
    // TODO 8: Update cart.updated_at timestamp
    // TODO 9: Commit transaction and return updated CartItem
    // Hints: Use SELECT FOR UPDATE to prevent race conditions
    // Handle unique constraint violations for concurrent additions
  }

  // Update cart item quantity with inventory validation
  async updateQuantity(sessionId, cartItemId, newQuantity) {
    // TODO 1: Validate newQuantity is positive integer
    // TODO 2: Start transaction and lock cart_item record
    // TODO 3: Verify cart_item belongs to session's cart
    // TODO 4: Get current product inventory level
    // TODO 5: Validate newQuantity doesn't exceed available inventory
    // TODO 6: If quantity is 0, delete cart_item (same as remove)
    // TODO 7: Otherwise update cart_item.quantity and timestamp
    // TODO 8: Update parent cart.updated_at timestamp
    // TODO 9: Commit transaction and return updated CartItem
```

```
  // Hint: Consider optimistic locking with version numbers
}

// Remove item completely from cart
async removeItem(sessionId, cartItemId) {
  // TODO 1: Start transaction for atomic removal
  // TODO 2: Verify cart_item exists and belongs to session's cart
  // TODO 3: Delete cart_item record from database
  // TODO 4: Update parent cart.updated_at timestamp
  // TODO 5: Check if cart is now empty and update status if needed
  // TODO 6: Commit transaction and return success boolean
  // Hint: Use CASCADE DELETE to handle orphaned references
}

// Retrieve complete cart contents with current pricing
async getCartContents(sessionId) {
  // TODO 1: Find active cart for session ID
  // TODO 2: Join cart_items with products to get current details
  // TODO 3: Compare price_snapshot with current product.price
  // TODO 4: Calculate line totals (quantity * price_snapshot)
  // TODO 5: Build CartSummary object with items and totals
  // TODO 6: Include price change flags for user notification
  // TODO 7: Return complete cart state for display
  // Hint: Use LEFT JOIN to handle deleted products gracefully
}

// Validate all cart items against current inventory and pricing
async validateInventory(sessionId) {
  // TODO 1: Retrieve all cart items for session
  // TODO 2: For each item, check current product inventory level
  // TODO 3: Compare requested quantity with available inventory
  // TODO 4: Check for significant price changes (> 5% threshold)
  // TODO 5: Identify discontinued or inactive products
```

```
    // TODO 6: Build ValidationReport with flagged items
    // TODO 7: Update cart item status flags for user attention
    // TODO 8: Return detailed validation results
    // Hint: Batch inventory checks to avoid N+1 query problems
  }
}

module.exports = CartService;
```

Cart Controller Implementation

HTTP Request Handlers (src/components/cart/CartController.js):

JAVASCRIPT

```javascript
class CartController {
  constructor(cartService) {
    this.cartService = cartService;
  }

  // POST /api/cart/items - Add item to cart
  async addItem(req, res) {
    // TODO 1: Extract and validate productId, quantity from request body
    // TODO 2: Get sessionId from req.session.cartId
    // TODO 3: Validate quantity is positive integer within reasonable limits
    // TODO 4: Call cartService.addItem with validated parameters
    // TODO 5: Handle inventory insufficient error with 409 status
    // TODO 6: Handle product not found error with 404 status
    // TODO 7: Return 201 Created with cart item details
    // TODO 8: Include updated cart totals in response
    // Hint: Validate quantity <= 99 to prevent abuse
  }

  // PUT /api/cart/items/:itemId - Update item quantity
  async updateQuantity(req, res) {
    // TODO 1: Extract cartItemId from route params, quantity from body
    // TODO 2: Validate quantity is non-negative integer
    // TODO 3: Get sessionId from request session
    // TODO 4: Call cartService.updateQuantity with parameters
    // TODO 5: Handle item not found with 404 status
    // TODO 6: Handle insufficient inventory with 409 status
    // TODO 7: Return 200 OK with updated item or deletion confirmation
    // Hint: Quantity 0 should remove item, return appropriate message
  }

  // GET /api/cart - Get complete cart contents
  async getCart(req, res) {
    // TODO 1: Extract sessionId from request session
```

```
    // TODO 2: Call cartService.getCartContents
    // TODO 3: Include price change notifications in response
    // TODO 4: Calculate and include cart totals (subtotal, tax, total)
    // TODO 5: Return 200 OK with complete cart summary
    // TODO 6: Handle empty cart case gracefully
    // Hint: Consider caching cart totals for performance
  }
}

module.exports = CartController;
```

Milestone Checkpoint

After implementing the cart component, verify functionality with these tests:

Test Commands:

```bash
# Run cart component tests
npm test -- --grep "Cart"

# Start development server
npm run dev

# Test cart operations via API
curl -X POST http://localhost:3000/api/cart/items \
  -H "Content-Type: application/json" \
  -d '{"productId": 1, "quantity": 2}'
```

Expected Behavior:

- Adding items returns 201 with cart item details and updated totals
- Inventory validation prevents adding more items than available stock
- Cart contents persist across browser refresh (session-based)
- Concurrent additions to same product update quantity correctly
- Price changes are detected and flagged in cart display

Debugging Checklist:

- Verify session middleware creates cart structure in req.session
- Check database foreign key constraints aren't causing silent failures
- Confirm inventory calculations account for items already in other carts
- Validate price_snapshot captures integer cents, not floating point dollars
- Test session expiry and cart cleanup work as configured

The cart component provides the foundation for user shopping sessions while maintaining data integrity and supporting the transition to checkout processing in Milestone 4.

User Authentication Component

> **Milestone(s):** Milestone 3 (User Authentication) - implements user registration, login, and profile management with secure password handling and session management

The user authentication component represents the security foundation of our e-commerce platform, establishing user identity and maintaining authenticated sessions across the shopping experience. This component must balance security requirements with user experience, ensuring that legitimate users can easily access their accounts while preventing unauthorized access to sensitive information like order history and payment details.

Authentication Mental Model: Building Security and ID Card System

Think of user authentication like the security system for a modern office building. When someone wants to work in the building, they first go through a **registration process** at the front desk - they provide their personal information, get their photo taken, and receive a temporary badge. The security office verifies their information, creates a permanent ID card with their photo and access credentials, and stores their information in the building's security database.

Each day when they arrive at work, they go through the **login process** - they present their ID card to the security scanner, which verifies the card is valid, hasn't expired, and belongs to them. If everything checks out, the security system grants them access and tracks that they're currently in the building. The security system maintains this **session state** throughout the day, knowing who's in the building and when they entered.

When they leave for the day, they badge out at the security desk, which **terminates their session** and records their departure. If they lose their ID card or it gets stolen, the security office can **invalidate** that card and issue a new one, ensuring the lost card can't be used by unauthorized people.

This physical security system maps directly to web authentication concepts. User registration creates credentials and stores them securely. Login validates those credentials and establishes a session. Session management tracks authenticated users and maintains their access state. Password security ensures that even if someone gains access to our credential storage, they can't easily misuse the information.

The key insight is that authentication is fundamentally about **identity verification** and **access control**. We must verify that users are who they claim to be, grant them appropriate access to resources, and maintain that access state securely throughout their session.

Authentication Interface

The authentication component exposes a clean interface that handles the complete user lifecycle from registration through logout. This interface must support both the immediate authentication needs and integrate seamlessly with the shopping cart and checkout components that depend on user identity.

Method Name	Parameters	Returns	Description
`register(email, password, confirmPassword)`	email: string, password: string, confirmPassword: string	`{success: boolean, userId?: number, errors?: string[]}`	Creates new user account with email and password validation, returns user ID on success or validation errors
`login(email, password)`	email: string, password: string	`{success: boolean, sessionId?: string, userId?: number, error?: string}`	Authenticates user credentials and creates session, returns session identifier and user ID on success
`logout(sessionId)`	sessionId: string	`{success: boolean}`	Terminates user session and clears authentication state, invalidating the session token
`validateSession(sessionId)`	sessionId: string	`{valid: boolean, userId?: number, expiresAt?: Date}`	Checks if session is valid and not expired, returns user ID and expiration if valid
`changePassword(userId, currentPassword, newPassword)`	userId: number, currentPassword: string, newPassword: string	`{success: boolean, error?: string}`	Updates user password after verifying current password, applies same security requirements as registration
`resetPasswordRequest(email)`	email: string	`{success: boolean, resetToken?: string}`	Initiates password reset flow, generates secure reset token (implementation stub for learning project)
`hashPassword(plaintext, saltRounds)`	plaintext: string, saltRounds: number	`Promise<string>`	Creates bcrypt hash of password with specified salt rounds, used internally for secure storage
`verifyPassword(plaintext, hashedPassword)`	plaintext: string, hashedPassword: string	`Promise<boolean>`	Compares plaintext password against stored hash using bcrypt, returns true if passwords match

The authentication interface follows a **session-based approach** rather than stateless tokens, which simplifies implementation for a learning project while still demonstrating core security principles. Each successful login creates a session record that tracks the user's authenticated state, and subsequent requests validate this session to determine access permissions.

Registration Flow Process:

1. The system receives registration request with email, password, and password confirmation
2. Email format validation ensures the address contains required components (local part, @ symbol, domain)
3. Password strength validation checks minimum length, character requirements, and common password blacklists

4. Password confirmation matching prevents typos that could lock users out of their accounts
5. Email uniqueness validation prevents duplicate accounts and ensures each email maps to exactly one user
6. Password hashing applies bcrypt with appropriate salt rounds to create secure storage format
7. User record creation persists the email and hashed password to the database with timestamps
8. Success response returns the new user ID while failure response includes specific validation errors

Login Flow Process:

1. The system receives login request with email and password credentials
2. User lookup retrieves the stored user record matching the provided email address
3. Password verification compares the provided plaintext against the stored bcrypt hash
4. Session creation generates a unique session identifier and stores session metadata
5. Session persistence saves session record with user ID, creation time, and expiration timestamp
6. Cookie configuration sets secure session cookie with appropriate flags and expiration
7. Success response returns session identifier and user ID for client-side state management

Password Security Algorithm

Password security represents the most critical aspect of user authentication, as compromised passwords can lead to unauthorized account access and data breaches. Our password security implementation follows industry best practices for hashing, salting, and validation while maintaining reasonable performance characteristics.

Password Hashing Procedure:

1. **Salt Generation**: The bcrypt algorithm automatically generates a unique random salt for each password hash operation. This salt prevents rainbow table attacks by ensuring that identical passwords produce different hash values across different users and different time periods.
2. **Cost Factor Selection**: We configure bcrypt with a cost factor (salt rounds) of 12, which provides strong security against brute-force attacks while maintaining acceptable performance for user login operations. This cost factor doubles the computation time with each increment, so cost factor 12 requires 4096 hash iterations.
3. **Hash Computation**: The bcrypt algorithm combines the plaintext password with the generated salt and applies the specified number of iteration rounds. The resulting hash includes the salt, cost factor, and hash value in a single encoded string format.
4. **Secure Storage**: The complete bcrypt hash string gets stored in the user record's password field. The hash format includes version information, cost parameters, salt, and hash value, enabling future verification operations without storing additional metadata.

Password Verification Procedure:

1. **Hash Extraction**: During login, the system retrieves the stored bcrypt hash from the user record matching the provided email address.
2. **Parameter Recovery**: The bcrypt library extracts the salt and cost factor from the stored hash format, ensuring verification uses the same parameters as the original hashing operation.
3. **Hash Comparison**: The system hashes the provided plaintext password using the extracted salt and cost factor, then compares the resulting hash against the stored hash value.
4. **Timing-Safe Comparison**: The bcrypt library performs constant-time comparison to prevent timing attacks that could leak information about hash values or password characteristics.

Password Strength Validation:

Requirement	Validation Rule	Error Message
Minimum Length	At least 8 characters	"Password must be at least 8 characters long"
Character Diversity	Contains uppercase, lowercase, and numeric characters	"Password must contain uppercase, lowercase, and numeric characters"
Special Characters	Contains at least one special character from approved set	"Password must contain at least one special character (!@#$%^&*)"
Common Password Check	Not found in common password blacklist	"Password is too common, please choose a different password"
Dictionary Word Check	Not a simple dictionary word or obvious pattern	"Password should not be a common dictionary word"
Personal Information	Doesn't contain email local part or obvious derivatives	"Password should not contain parts of your email address"

The password strength validation runs during registration and password change operations, providing immediate feedback to users about password requirements. This validation occurs on the server side to ensure security requirements are enforced regardless of client-side validation state.

> **Security Insight**: The combination of bcrypt hashing with proper salt rounds and comprehensive password strength validation creates multiple layers of protection. Even if an attacker gains access to our password database, the computational cost of cracking well-hashed passwords makes brute-force attacks impractical.

Authentication Architecture Decisions

The authentication component requires several critical architectural decisions that impact both security posture and system complexity. Each decision involves trade-offs between security, performance, scalability, and implementation complexity.

> **Decision: Session-Based vs Token-Based Authentication**
>
> - **Context**: Web applications can maintain user authentication state through server-side sessions or client-side tokens, each with distinct security and scalability implications
> - **Options Considered**: Server-side sessions with cookies, JWT tokens, hybrid approach with refresh tokens
> - **Decision**: Server-side sessions with secure HTTP cookies
> - **Rationale**: Sessions provide better security for a learning project by keeping authentication state server-side, enable simple session invalidation, and integrate naturally with Express middleware patterns
> - **Consequences**: Requires server-side session storage and limits horizontal scaling, but provides stronger security guarantees and simpler revocation mechanisms

Authentication Approach	Pros	Cons	Chosen?
Server-Side Sessions	Simple revocation, secure state storage, integrated middleware	Requires server storage, scaling complexity	✅ Yes
JWT Tokens	Stateless, scalable, standard format	Difficult revocation, client-side storage risks	❌ No
Hybrid (JWT + Refresh)	Balance of security and scalability	Implementation complexity, multiple token types	❌ No

Decision: Password Hashing Algorithm Selection

- **Context**: Password storage requires irreversible hashing with salt to protect against database compromise and rainbow table attacks
- **Options Considered**: bcrypt, Argon2, PBKDF2, scrypt
- **Decision**: bcrypt with cost factor 12
- **Rationale**: bcrypt provides proven security with widespread library support, reasonable performance characteristics, and automatic salt handling that reduces implementation errors
- **Consequences**: Strong protection against offline attacks with configurable work factor, but requires careful cost factor tuning for performance

Hashing Algorithm	Pros	Cons	Chosen?
bcrypt	Proven track record, automatic salt handling, configurable cost	Older algorithm, limited memory usage	✅ Yes
Argon2	Modern design, memory-hard function, winner of password hashing competition	Less widespread support, more complex configuration	❌ No
PBKDF2	Standards-based, configurable iterations	Vulnerable to hardware attacks, no memory hardness	❌ No

Decision: Session Storage Strategy

- **Context**: Session data must persist across HTTP requests and survive server restarts while maintaining reasonable performance
- **Options Considered**: In-memory storage, database storage, Redis cache, file-based storage
- **Decision**: Database storage with in-memory caching
- **Rationale**: Database storage ensures session persistence across server restarts while in-memory caching provides fast session validation for authenticated requests
- **Consequences**: Adds database table and queries for session operations, but provides durability and reasonable performance for a learning project

Decision: Session Expiration Strategy

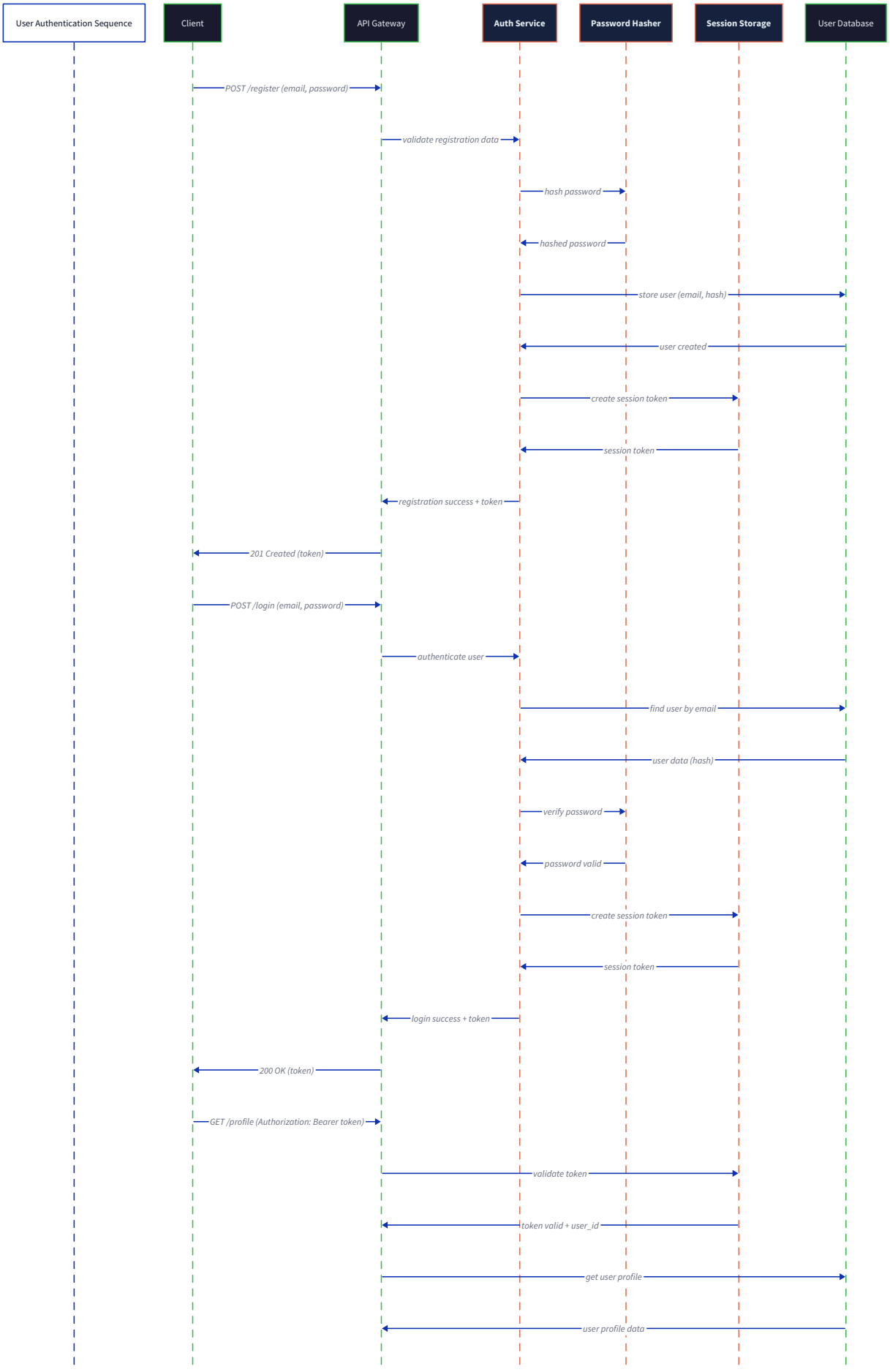
- **Context**: Sessions must eventually expire to limit exposure from compromised session tokens and abandoned sessions
- **Options Considered**: Fixed expiration time, sliding expiration, idle timeout, absolute maximum lifetime
- **Decision**: Sliding expiration with 24-hour timeout and 7-day maximum lifetime
- **Rationale**: Sliding expiration maintains convenience for active users while idle timeout protects against abandoned sessions, and maximum lifetime limits long-term exposure
- **Consequences**: Requires session timestamp updates on each request and periodic cleanup of expired sessions

The session expiration implementation tracks both last activity time and session creation time. Each authenticated request updates the last activity timestamp, extending the session lifetime by the idle timeout period. However, sessions cannot extend beyond the absolute maximum lifetime, forcing periodic re-authentication even for highly active users.

Session Cookie Configuration:

Cookie Attribute	Value	Purpose
`httpOnly`	`true`	Prevents JavaScript access to session cookie, protecting against XSS attacks
`secure`	`true` (production)	Ensures cookie transmission only over HTTPS connections
`sameSite`	`'strict'`	Prevents CSRF attacks by blocking cross-site cookie transmission
`maxAge`	`24 * 60 * 60 * 1000`	Sets cookie expiration to 24 hours from creation
`signed`	`true`	Enables cookie signature verification to prevent tampering

The cookie configuration provides defense-in-depth against common web security vulnerabilities. HTTP-only cookies prevent client-side JavaScript from accessing session tokens, even in the presence of XSS vulnerabilities. Secure cookies ensure session tokens only transmit over encrypted connections, preventing network eavesdropping. Same-site restrictions prevent CSRF attacks that attempt to use victim session tokens from malicious websites.


User Authentication Sequence
Client
API Gateway
Auth Service
Password Hasher
Session Storage
User Database
POST /register (email, password)
validate registration data
hash password
hashed password
store user (email, hash)
user created
create session token
session token
registration success + token
201 Created (token)
POST /login (email, password)
authenticate user
find user by email
user data (hash)
verify password
password valid
create session token
session token
login success + token
200 OK (token)
GET /profile (Authorization: Bearer token)
validate token
token valid + user_id
get user profile
user profile data

Common Authentication Pitfalls

Authentication implementations commonly suffer from security vulnerabilities and usability issues that can compromise user accounts or degrade the user experience. Understanding these pitfalls helps developers build more secure and robust authentication systems.

⚠️ Pitfall: Timing Attacks on Login Validation

Login validation can inadvertently leak information about user account existence through timing differences between valid and invalid email addresses. When the system performs expensive password hashing only for existing accounts, attackers can measure response times to determine which email addresses have registered accounts.

Why it's problematic: Attackers can enumerate valid user accounts by measuring login attempt response times, then focus password attacks on confirmed accounts rather than guessing both email addresses and passwords.

Detection: Login responses for non-existent accounts return significantly faster than responses for existing accounts with incorrect passwords.

Fix: Always perform password hashing during login validation, even for non-existent accounts. Use a dummy hash operation with the same computational cost as legitimate verification to normalize response times.

⚠️ Pitfall: Weak Password Requirements

Implementing minimal password requirements like "8 characters minimum" provides false security while frustrating users with arbitrary restrictions that don't meaningfully improve security.

Why it's problematic: Short passwords with predictable patterns remain vulnerable to dictionary attacks and brute-force attempts, while overly complex requirements lead to password reuse and written passwords.

Detection: Password cracking tools successfully compromise user accounts despite meeting stated requirements, or user support receives frequent password reset requests.

Fix: Implement comprehensive password validation including length requirements, character diversity, common password blacklists, and pattern detection. Focus on entropy rather than arbitrary character requirements.

⚠️ Pitfall: Session Fixation Vulnerabilities

Reusing session identifiers across authentication state changes allows attackers to hijack user sessions by tricking victims into authenticating with a known session ID.

Why it's problematic: Attackers can provide victims with specific session tokens, then gain authenticated access when victims log into their accounts using those predetermined sessions.

Detection: Session tokens remain unchanged after login, or session management doesn't properly invalidate pre-authentication sessions.

Fix: Generate new session identifiers immediately after successful authentication, invalidating any pre-existing session data and ensuring authenticated sessions use fresh tokens.

⚠️ Pitfall: Insecure Password Reset Implementation

Password reset functionality often introduces security vulnerabilities through predictable reset tokens, overly long token lifetimes, or insufficient validation during the reset process.

Why it's problematic: Weak reset tokens enable account takeover attacks, while poorly designed reset flows can bypass normal authentication security measures.

Detection: Password reset tokens use predictable formats, remain valid indefinitely, or allow multiple concurrent reset attempts without proper controls.

Fix: Generate cryptographically secure reset tokens with limited lifetimes, invalidate tokens after use or timeout, and require additional verification steps for sensitive account changes.

⚠️ Pitfall: Session Management Memory Leaks

Server-side session storage can accumulate expired sessions over time, leading to memory exhaustion and performance degradation as the application processes increasing numbers of abandoned sessions.

Why it's problematic: Abandoned sessions consume server memory indefinitely, while session validation becomes slower as the session store grows with expired entries.

Detection: Server memory usage increases over time without corresponding user activity growth, or session validation performance degrades as the application runs longer.

Fix: Implement automatic session cleanup that periodically removes expired sessions, and set reasonable session lifetime limits to prevent indefinite accumulation.

⚠️ Pitfall: Insufficient Session Validation

Incomplete session validation allows requests with expired, invalid, or tampered session tokens to access protected resources, undermining the authentication system's security guarantees.

Why it's problematic: Attackers can potentially access user accounts using old session tokens, forged tokens, or sessions from compromised accounts that should have been invalidated.

Detection: Protected endpoints accept requests with expired sessions, invalid session formats, or sessions that don't correspond to valid user accounts.

Fix: Implement comprehensive session validation that checks token format, expiration times, user account status, and session integrity on every protected request.

The authentication component's security depends on careful attention to these implementation details. Each pitfall represents a common mistake that can compromise the entire system's security posture, making thorough testing and security review essential for production deployments.

Implementation Guidance

The authentication component requires careful implementation to balance security requirements with development complexity. This guidance provides complete starter code for infrastructure components and detailed skeletons for core authentication logic.

A. Technology Recommendations:

Component	Simple Option	Advanced Option
Password Hashing	`bcrypt` library with default settings	`argon2` with tuned memory and time parameters
Session Storage	Database sessions with in-memory cache	Redis with session clustering
Input Validation	Manual regex and length checks	`joi` or `express-validator` middleware
Security Headers	Manual header setting	`helmet` middleware package
Rate Limiting	Simple in-memory counter	`express-rate-limit` with Redis backend

B. Recommended File Structure:

```
src/
  auth/
    auth-controller.js       ← HTTP route handlers for registration/login
    auth-service.js          ← Core authentication business logic
    password-service.js      ← Password hashing and validation utilities
    session-manager.js       ← Session creation and validation
    auth-middleware.js       ← Express middleware for protected routes
    auth-validator.js        ← Input validation for auth requests
  models/
    User.js                  ← User entity with database operations
    Session.js               ← Session entity with expiration logic
  middleware/
    security.js              ← Security headers and CSRF protection
```

C. Infrastructure Starter Code:

Password Service (`src/auth/password-service.js`):

JAVASCRIPT

```javascript
const bcrypt = require('bcrypt');

const SALT_ROUNDS = 12;
const MIN_PASSWORD_LENGTH = 8;

// Common passwords to reject during validation
const COMMON_PASSWORDS = [
    'password', '123456', '123456789', 'qwerty', 'abc123',
    'password123', 'admin', 'letmein', 'welcome', 'monkey'
];

class PasswordService {
    /**
     * Hashes a plaintext password using bcrypt with secure salt rounds
     * @param {string} plaintextPassword - The password to hash
     * @returns {Promise<string>} - The bcrypt hash string
     */
    async hashPassword(plaintextPassword) {
        try {
            const hash = await bcrypt.hash(plaintextPassword, SALT_ROUNDS);
            return hash;
        } catch (error) {
            throw new Error(`Password hashing failed: ${error.message}`);
        }
    }

    /**
     * Verifies a plaintext password against a stored bcrypt hash
     * @param {string} plaintextPassword - The password to verify
     * @param {string} hashedPassword - The stored bcrypt hash
     * @returns {Promise<boolean>} - True if password matches
     */
    async verifyPassword(plaintextPassword, hashedPassword) {
        try {
```

```
            const isMatch = await bcrypt.compare(plaintextPassword, hashedPassword);
            return isMatch;
        } catch (error) {
            // Always perform a dummy hash operation to prevent timing attacks
            await bcrypt.hash('dummy', SALT_ROUNDS);
            return false;
        }
    }

    /**
     * Validates password strength according to security requirements
     * @param {string} password - The password to validate
     * @param {string} email - User's email to check for personal information
     * @returns {Object} - {isValid: boolean, errors: string[]}
     */
    validatePasswordStrength(password, email = '') {
        const errors = [];

        if (!password || password.length < MIN_PASSWORD_LENGTH) {
            errors.push(`Password must be at least ${MIN_PASSWORD_LENGTH} characters long`);
        }

        if (!/[a-z]/.test(password)) {
            errors.push('Password must contain at least one lowercase letter');
        }

        if (!/[A-Z]/.test(password)) {
            errors.push('Password must contain at least one uppercase letter');
        }

        if (!/\d/.test(password)) {
            errors.push('Password must contain at least one number');
        }

        if (!/[!@#$%^&*()_+\-=\[\]{};':"\\|,.<>\/?]/.test(password)) {
```

```
            errors.push('Password must contain at least one special character');
        }

        if (COMMON_PASSWORDS.includes(password.toLowerCase())) {
            errors.push('Password is too common, please choose a different password');
        }

        if (email) {
            const emailLocal = email.split('@')[0].toLowerCase();
            if (password.toLowerCase().includes(emailLocal)) {
                errors.push('Password should not contain parts of your email address');
            }
        }

        return {
            isValid: errors.length === 0,
            errors: errors
        };
    }
}

module.exports = new PasswordService();
```

Session Manager (`src/auth/session-manager.js`):

JAVASCRIPT

```javascript
const crypto = require('crypto');

const SESSION_TIMEOUT_MS = 24 * 60 * 60 * 1000; // 24 hours
const MAX_SESSION_LIFETIME_MS = 7 * 24 * 60 * 60 * 1000; // 7 days
const CLEANUP_INTERVAL_MS = 60 * 60 * 1000; // 1 hour

class SessionManager {
    constructor(database) {
        this.db = database;
        this.sessionCache = new Map();

        // Start periodic cleanup of expired sessions
        this.cleanupInterval = setInterval(() => {
            this.cleanupExpiredSessions();
        }, CLEANUP_INTERVAL_MS);
    }

    /**
     * Generates a cryptographically secure session ID
     * @returns {string} - Random session identifier
     */
    generateSessionId() {
        return crypto.randomBytes(32).toString('hex');
    }

    /**
     * Creates a new session for an authenticated user
     * @param {number} userId - The authenticated user's ID
     * @returns {Promise<string>} - The new session ID
     */
    async createSession(userId) {
        const sessionId = this.generateSessionId();
        const now = new Date();
        const expiresAt = new Date(now.getTime() + SESSION_TIMEOUT_MS);
```

```
        const maxLifetime = new Date(now.getTime() + MAX_SESSION_LIFETIME_MS);

        const sessionData = {
            session_id: sessionId,
            user_id: userId,
            created_at: now,
            last_activity: now,
            expires_at: expiresAt,
            max_lifetime: maxLifetime
        };

        // Store in database
        await this.db.run(
            `INSERT INTO sessions (session_id, user_id, created_at, last_activity, expires_at, max_lifetime)
             VALUES (?, ?, ?, ?, ?, ?)`,
            [sessionId, userId, now, now, expiresAt, maxLifetime]
        );

        // Cache for fast access
        this.sessionCache.set(sessionId, sessionData);

        return sessionId;
    }

    /**
     * Removes expired sessions from database and cache
     */
    async cleanupExpiredSessions() {
        const now = new Date();

        // Remove from database
        await this.db.run(
            'DELETE FROM sessions WHERE expires_at < ? OR max_lifetime < ?',
            [now, now]
```

```
        );

        // Remove from cache
        for (const [sessionId, session] of this.sessionCache.entries()) {
            if (session.expires_at < now || session.max_lifetime < now) {
                this.sessionCache.delete(sessionId);
            }
        }
    }

    /**
     * Validates a session and updates last activity time
     * @param {string} sessionId - The session ID to validate
     * @returns {Promise<Object|null>} - Session data if valid, null if invalid
     */
    async validateSession(sessionId) {
        // Check cache first
        let session = this.sessionCache.get(sessionId);

        if (!session) {
            // Load from database
            session = await this.db.get(
                'SELECT * FROM sessions WHERE session_id = ?',
                [sessionId]
            );

            if (session) {
                // Convert date strings back to Date objects
                session.created_at = new Date(session.created_at);
                session.last_activity = new Date(session.last_activity);
                session.expires_at = new Date(session.expires_at);
                session.max_lifetime = new Date(session.max_lifetime);
```

```
            this.sessionCache.set(sessionId, session);
        }
    }

    if (!session) {
        return null;
    }

    const now = new Date();

    // Check if session has expired
    if (session.expires_at < now || session.max_lifetime < now) {
        await this.destroySession(sessionId);
        return null;
    }

    // Update last activity and extend expiration
    session.last_activity = now;
    session.expires_at = new Date(now.getTime() + SESSION_TIMEOUT_MS);

    // Don't extend beyond max lifetime
    if (session.expires_at > session.max_lifetime) {
        session.expires_at = session.max_lifetime;
    }

    // Update in database and cache
    await this.db.run(
        'UPDATE sessions SET last_activity = ?, expires_at = ? WHERE session_id = ?',
        [session.last_activity, session.expires_at, sessionId]
    );

    this.sessionCache.set(sessionId, session);

    return {
```

```
            userId: session.user_id,
            createdAt: session.created_at,
            lastActivity: session.last_activity,
            expiresAt: session.expires_at
        };
    }

    /**
     * Destroys a session, removing it from both database and cache
     * @param {string} sessionId - The session ID to destroy
     */
    async destroySession(sessionId) {
        await this.db.run('DELETE FROM sessions WHERE session_id = ?', [sessionId]);
        this.sessionCache.delete(sessionId);
    }

    /**
     * Closes the session manager and stops cleanup processes
     */
    close() {
        if (this.cleanupInterval) {
            clearInterval(this.cleanupInterval);
        }
    }
}

module.exports = SessionManager;
```

D. Core Logic Skeleton Code:

Authentication Service (`src/auth/auth-service.js`):

JAVASCRIPT

```javascript
const passwordService = require('./password-service');

class AuthService {
    constructor(database, sessionManager) {
        this.db = database;
        this.sessions = sessionManager;
    }

    /**
     * Registers a new user account with email and password validation
     * @param {string} email - User's email address
     * @param {string} password - User's chosen password
     * @param {string} confirmPassword - Password confirmation
     * @returns {Promise<Object>} - {success: boolean, userId?: number, errors?: string[]}
     */
    async register(email, password, confirmPassword) {
        // TODO 1: Validate email format using regex or email validation library
        // TODO 2: Check if email is already registered by querying users table
        // TODO 3: Validate password matches confirmPassword exactly
        // TODO 4: Use passwordService.validatePasswordStrength() to check password requirements
        // TODO 5: Hash the password using passwordService.hashPassword()
        // TODO 6: Insert new user record with email and hashed password
        // TODO 7: Return success response with new user ID or validation errors
        // Hint: Use transactions to ensure atomic user creation
    }

    /**
     * Authenticates user credentials and creates a session
     * @param {string} email - User's email address
     * @param {string} password - User's password
     * @returns {Promise<Object>} - {success: boolean, sessionId?: string, userId?: number, error?: string}
     */
    async login(email, password) {
```

```
        // TODO 1: Query database for user record matching the email address
        // TODO 2: If user not found, perform dummy hash operation to prevent timing attacks
        // TODO 3: Use passwordService.verifyPassword() to check password against stored hash
        // TODO 4: If password verification fails, return authentication error
        // TODO 5: Create new session using sessionManager.createSession(userId)
        // TODO 6: Return success response with session ID and user ID
        // Hint: Always perform password hashing even for non-existent users
    }

    /**
     * Terminates user session and clears authentication state
     * @param {string} sessionId - The session to terminate
     * @returns {Promise<Object>} - {success: boolean}
     */
    async logout(sessionId) {
        // TODO 1: Validate that sessionId is provided and properly formatted
        // TODO 2: Use sessionManager.destroySession() to remove session
        // TODO 3: Return success confirmation
        // Hint: Logout should succeed even for invalid session IDs
    }

    /**
     * Changes user password after verifying current password
     * @param {number} userId - The user whose password to change
     * @param {string} currentPassword - User's current password
     * @param {string} newPassword - User's new password
     * @returns {Promise<Object>} - {success: boolean, error?: string}
     */
    async changePassword(userId, currentPassword, newPassword) {
        // TODO 1: Retrieve user record from database using userId
        // TODO 2: Verify currentPassword matches stored hash using passwordService
        // TODO 3: Validate newPassword strength using passwordService.validatePasswordStrength()
        // TODO 4: Hash newPassword using passwordService.hashPassword()
```

```
        // TODO 5: Update user record with new password hash
        // TODO 6: Invalidate all existing sessions for this user (security measure)
        // TODO 7: Return success confirmation or specific error messages
    }
}

module.exports = AuthService;
```

Authentication Middleware (`src/auth/auth-middleware.js`):

JAVASCRIPT

```javascript
/**
 * Express middleware that enforces authentication for protected routes
 * Validates session and populates req.user with authenticated user information
 */
function requireAuth(sessionManager) {
    return async (req, res, next) => {
        // TODO 1: Extract session ID from signed cookie using req.signedCookies
        // TODO 2: If no session cookie exists, return 401 Unauthorized error
        // TODO 3: Use sessionManager.validateSession() to check session validity
        // TODO 4: If session is invalid or expired, clear cookie and return 401 error
        // TODO 5: Populate req.user with {userId, sessionId} for downstream middleware
        // TODO 6: Call next() to continue to the protected route handler
        // Hint: Set appropriate WWW-Authenticate header for 401 responses
    };
}

/**
 * Express middleware that loads user information if authenticated but doesn't require it
 * Used for routes that work for both authenticated and anonymous users
 */
function optionalAuth(sessionManager) {
    return async (req, res, next) => {
        // TODO 1: Extract session ID from signed cookie, but don't fail if missing
        // TODO 2: If session exists, validate it using sessionManager.validateSession()
        // TODO 3: If valid, populate req.user with authentication information
        // TODO 4: If invalid or missing, leave req.user undefined
        // TODO 5: Always call next() regardless of authentication state
        // Hint: This middleware enables cart transfer from anonymous to authenticated sessions
    };
}

module.exports = {
    requireAuth,
```

```
    optionalAuth
};
```

E. Language-Specific Hints:

- **Session Cookies**: Use `express-session` or manual cookie handling with `res.cookie()` and `req.signedCookies` for secure session management
- **Password Hashing**: Install `bcrypt` package and use async methods (`bcrypt.hash` , `bcrypt.compare`) to avoid blocking the event loop
- **Timing Attacks**: Always perform password hashing operations even for non-existent users using a dummy hash to normalize response times
- **Database Transactions**: Use `db.run()` with transaction wrapper for atomic user creation and session management
- **Input Validation**: Sanitize email inputs and use parameterized queries to prevent SQL injection attacks
- **Error Handling**: Return consistent error response formats and avoid leaking sensitive information in error messages

F. Milestone Checkpoint:

After implementing the authentication component, verify the following behavior:

Registration Testing:

```bash
# Test user registration endpoint
curl -X POST http://localhost:3000/auth/register \
  -H "Content-Type: application/json" \
  -d '{"email":"test@example.com","password":"SecurePass123!","confirmPassword":"SecurePass123!"}'

# Expected: {"success": true, "userId": 1}
```

Login Testing:

```bash
# Test user login endpoint
curl -X POST http://localhost:3000/auth/login \
  -H "Content-Type: application/json" \
  -c cookies.txt \
  -d '{"email":"test@example.com","password":"SecurePass123!"}'

# Expected: {"success": true, "sessionId": "...", "userId": 1}
# Should set secure session cookie
```

Protected Route Testing:

```bash
# Test accessing protected route with session cookie
curl -X GET http://localhost:3000/api/profile \
  -H "Content-Type: application/json" \
  -b cookies.txt

# Expected: User profile data or authentication required error
```

Signs of Problems:

- Registration succeeds with weak passwords → Check password validation implementation
- Login responses vary significantly in timing → Implement timing attack protection
- Sessions don't persist across requests → Verify cookie configuration and session storage
- Multiple registrations allowed with same email → Add email uniqueness constraint

Checkout Process Component

> **Milestone(s):** Milestone 4 (Checkout Process) - implements checkout flow with order creation, including address collection, order summary display, inventory validation, and payment processing integration

The checkout process represents the culmination of the e-commerce user journey, where browsing and cart management transform into committed purchases. This component orchestrates the complex sequence of operations required to convert a shopping cart into a confirmed order while maintaining inventory consistency and handling payment processing. The checkout process must coordinate multiple system components, validate business rules, and ensure transactional integrity across potentially distributed operations.

Checkout Mental Model: Cashier Transaction Analogy

Think of the checkout process like a cashier transaction at a physical store. When you approach the checkout counter with your shopping cart, the cashier follows a specific sequence of steps that mirror our digital checkout process. First, they scan each item to verify it's still available and hasn't changed price since you picked it up - this corresponds to our inventory validation step. Next, they collect your payment information and shipping address, similar to our address collection phase. The cashier then processes your payment, which maps to our payment gateway integration. Finally, they print a receipt and update the store's inventory system, equivalent to our order confirmation and inventory adjustment operations.

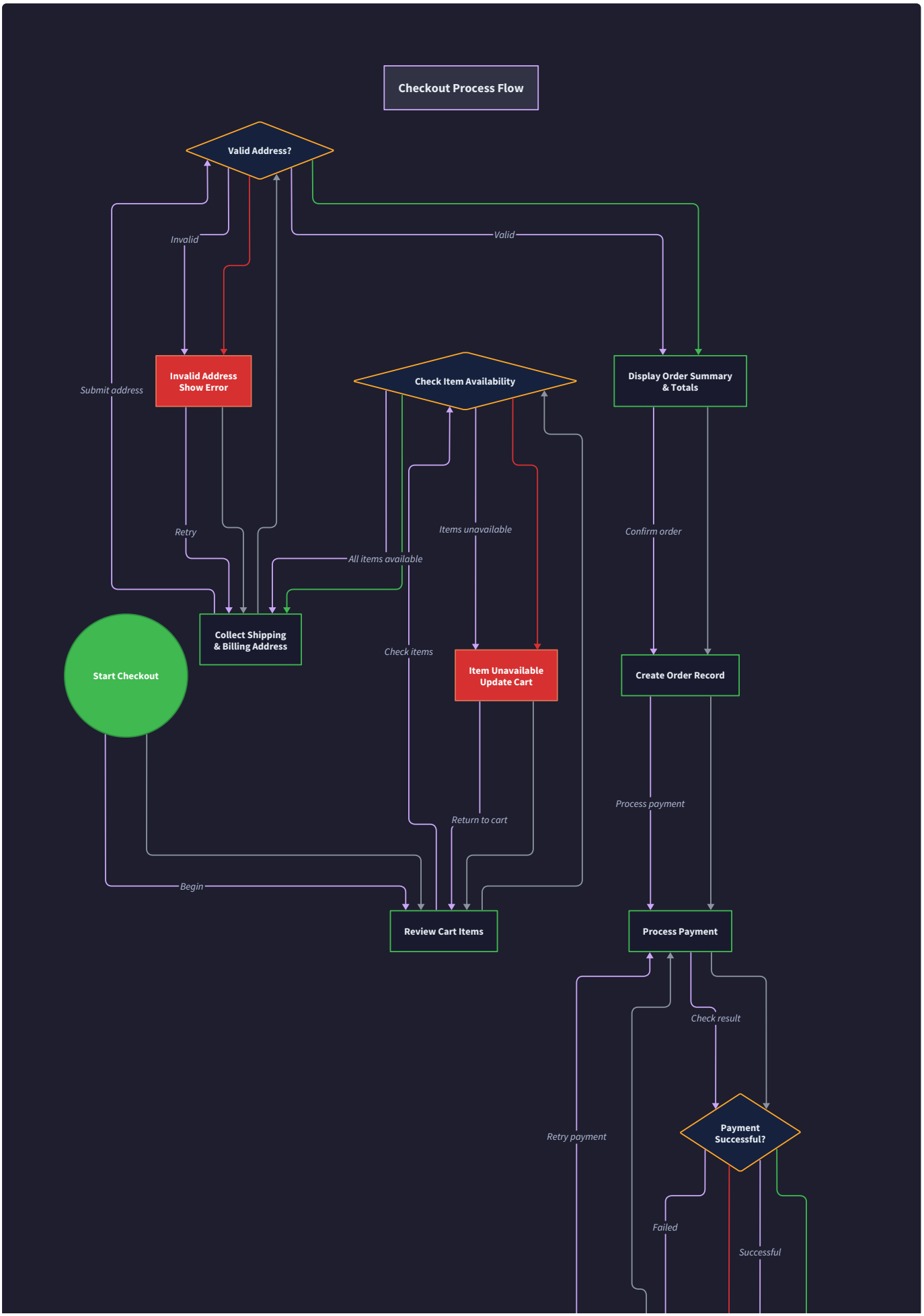
This mental model helps us understand why checkout must be an atomic operation - just as a physical store wouldn't want to charge your credit card but forget to give you the merchandise, our system must ensure that payment, inventory updates, and order creation all succeed together or fail together. The cashier analogy also illuminates why we need to handle interruptions gracefully - if the credit card machine breaks mid-transaction, the cashier needs clear procedures to either complete the sale through alternative means or void the transaction entirely.

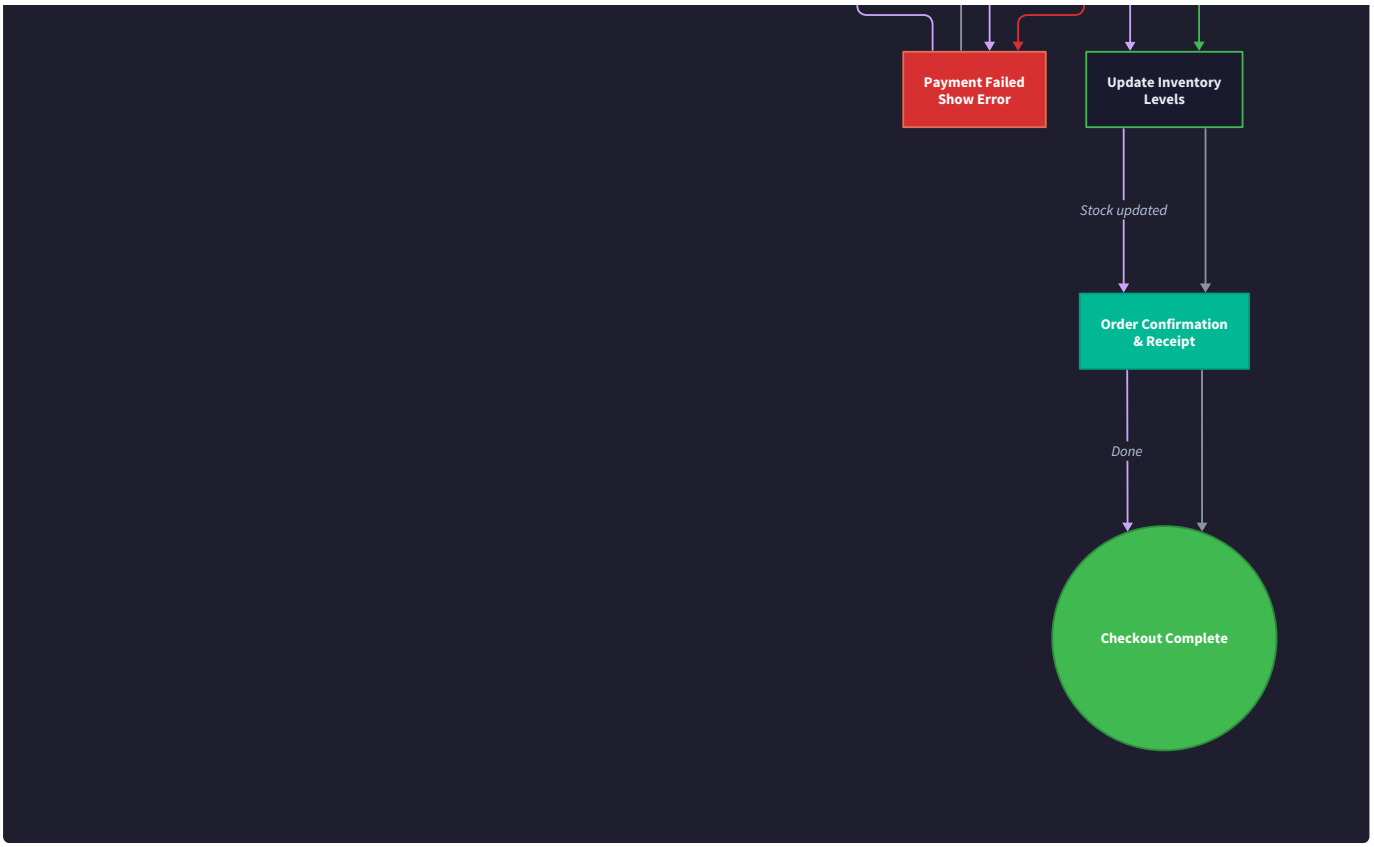
The key insight from this analogy is that checkout represents a **state transition boundary** where tentative actions (items in cart) become committed actions (purchased items). Just as the cashier controls this transition point and ensures it happens correctly, our checkout component must orchestrate this critical transformation while handling the various failure modes that can occur.

Checkout Interface

The checkout interface defines the API endpoints that guide users through the multi-step process of converting their shopping cart into a confirmed order. These endpoints follow a stateful progression where each step builds upon the previous one and validates

the current state before proceeding.


Checkout Process Flow
Valid Address?
Invalid
Valid
Invalid Address Show Error
Submit address
Check Item Availability
Display Order Summary & Totals
Retry
All items available
Items unavailable
Confirm order
Collect Shipping & Billing Address
Start Checkout
Check items
Item Unavailable Update Cart
Create Order Record
Process payment
Return to cart
Begin
Review Cart Items
Process Payment
Check result
Retry payment
Payment Successful?
Failed
Successful



Checkout API Endpoints

Method	Endpoint	Parameters	Returns	Description
`POST`	`/api/checkout/start`	`sessionId: string`	`CheckoutSession`	Initiates checkout process and validates cart contents
`POST`	`/api/checkout/address`	`sessionId: string, shippingAddress: Address, billingAddress?: Address`	`CheckoutSession`	Collects and validates shipping address information
`GET`	`/api/checkout/summary`	`sessionId: string`	`OrderSummary`	Returns complete order preview with pricing breakdown
`POST`	`/api/checkout/validate`	`sessionId: string`	`ValidationReport`	Performs final inventory and price validation
`POST`	`/api/checkout/confirm`	`sessionId: string, paymentMethod: PaymentMethod`	`OrderConfirmation`	Creates order and processes payment atomically

Method	Endpoint	Parameters	Returns	Description
`GET`	`/api/checkout/status/{checkoutId}`	`checkoutId: string`	`CheckoutStatus`	Retrieves current status of checkout session

Checkout Data Structures

The checkout process relies on several data structures that capture the state and progression of the order creation workflow:

Structure	Field	Type	Description
`CheckoutSession`	`id`	`string`	Unique identifier for checkout session
	`sessionId`	`string`	Reference to user's cart session
	`status`	`CheckoutStatus`	Current stage of checkout process
	`cartSnapshot`	`CartItem[]`	Frozen copy of cart at checkout start
	`shippingAddress`	`Address`	Customer's shipping information
	`billingAddress`	`Address`	Customer's billing information
	`priceTotal`	`number`	Total price in cents including taxes
	`createdAt`	`Date`	Timestamp when checkout session began
	`expiresAt`	`Date`	Expiration time for checkout session
`Address`	`fullName`	`string`	Recipient's complete name
	`addressLine1`	`string`	Primary street address
	`addressLine2`	`string`	Secondary address information (optional)
	`city`	`string`	City name
	`state`	`string`	State or province code
	`postalCode`	`string`	ZIP or postal code
	`country`	`string`	ISO country code
	`phoneNumber`	`string`	Contact phone number
`OrderSummary`	`items`	`OrderItem[]`	List of products being purchased
	`subtotal`	`number`	Sum of item prices in cents
	`shipping`	`number`	Shipping cost in cents
	`tax`	`number`	Tax amount in cents
	`total`	`number`	Final total amount in cents
	`shippingAddress`	`Address`	Delivery destination
	`estimatedDelivery`	`Date`	Expected delivery date

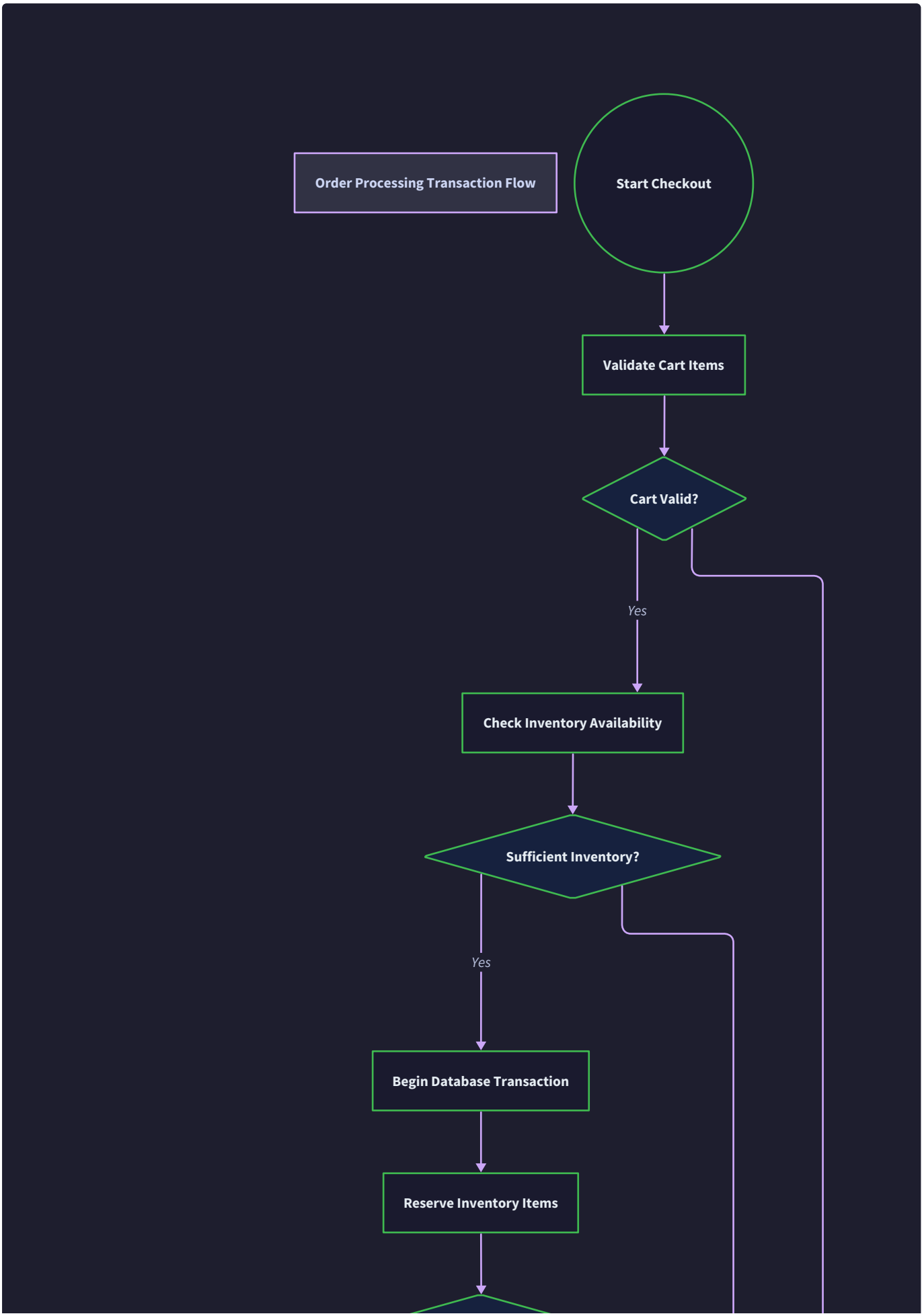
Address Validation Requirements

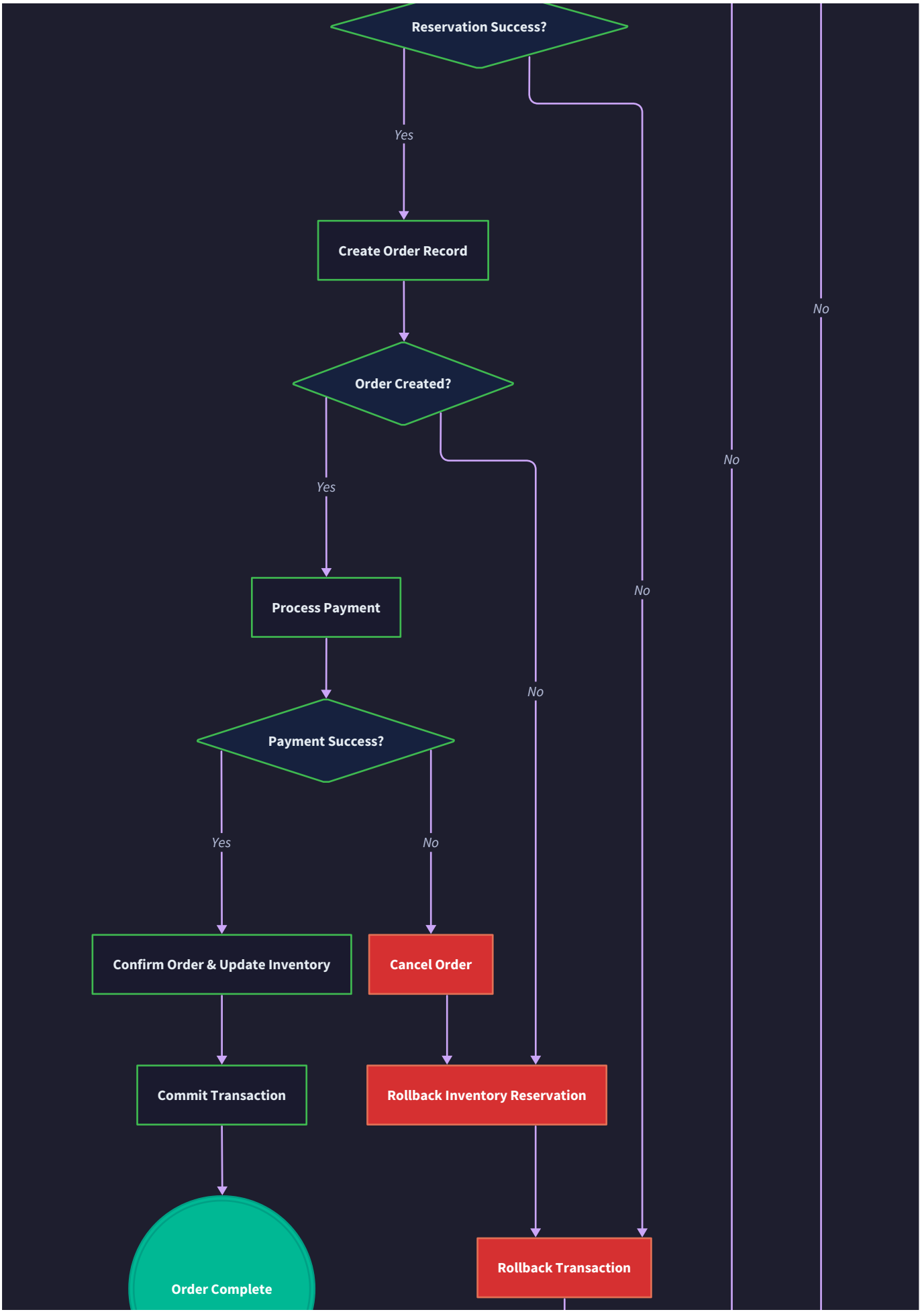
Address collection must implement comprehensive validation to ensure successful order fulfillment. The validation rules enforce both format requirements and business constraints:

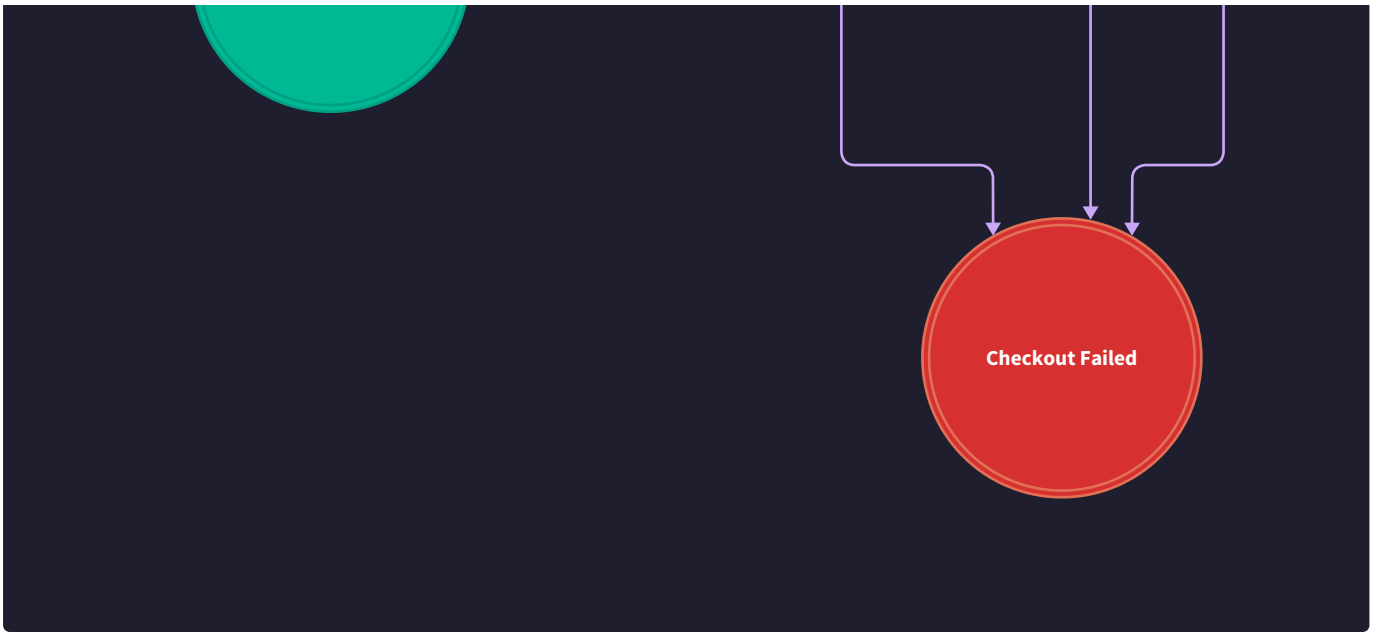
Field	Validation Rule	Error Message
`fullName`	Required, 2-100 characters, no special characters except hyphens and apostrophes	"Name must be 2-100 characters with letters, spaces, hyphens, and apostrophes only"
`addressLine1`	Required, 5-100 characters, alphanumeric with standard punctuation	"Street address must be 5-100 characters"
`city`	Required, 2-50 characters, letters and spaces only	"City must be 2-50 characters with letters and spaces only"
`state`	Required, valid state/province code for country	"Please select a valid state or province"
`postalCode`	Required, format validated against country-specific patterns	"Please enter a valid postal code for your country"
`phoneNumber`	Required, valid phone format for country	"Please enter a valid phone number"

Order Processing Algorithm

The order processing algorithm orchestrates the conversion of a shopping cart into a confirmed order through a series of atomic operations. This algorithm must handle the complexity of coordinating inventory updates, payment processing, and order creation while providing strong consistency guarantees.


Order Processing Transaction Flow
Start Checkout
Validate Cart Items
Cart Valid?
Yes
Check Inventory Availability
Sufficient Inventory?
Yes
Begin Database Transaction
Reserve Inventory Items


Reservation Success?
Yes
No
Create Order Record
Order Created?
Yes
No
Process Payment
Payment Success?
Yes
No
Confirm Order & Update Inventory
Cancel Order
Commit Transaction
Rollback Inventory Reservation
Order Complete
Rollback Transaction
No
No
No



Transaction Isolation Strategy

The checkout process implements **pessimistic locking** with **transaction isolation** to prevent race conditions during order creation. This approach ensures that inventory levels remain consistent even when multiple customers attempt to purchase the same products simultaneously.

Step-by-Step Order Processing Flow

1. **Checkout Session Initialization**: The system creates a new `CheckoutSession` record linked to the user's cart session. This session captures a frozen snapshot of the cart contents at the moment checkout begins, preventing modifications during the checkout process. The session includes an expiration timestamp (typically 15 minutes) to prevent indefinite reservation of inventory.

2. **Cart Content Validation**: The system validates each item in the cart snapshot against current product data. This validation checks that products are still active, prices haven't changed significantly, and categories haven't been modified. Any discrepancies result in a validation report that the user must acknowledge before proceeding.

3. **Inventory Availability Check**: For each product in the checkout session, the system performs a SELECT FOR UPDATE query to lock the product record and verify sufficient inventory. This pessimistic locking approach prevents other transactions from modifying inventory levels during the checkout process. The system calculates the new inventory level but doesn't commit the change yet.

4. **Address Validation and Geocoding**: The provided shipping address undergoes format validation followed by verification through a geocoding service. This step ensures the address is deliverable and calculates accurate shipping costs. Invalid addresses halt the process with specific error messages indicating which fields need correction.

5. **Shipping and Tax Calculation**: Based on the validated address and cart contents, the system calculates shipping costs and applicable taxes. This calculation considers product weight, dimensions, shipping zone, and local tax regulations. The total price is computed and stored in the checkout session for consistency.

6. **Payment Authorization**: The system submits a payment authorization request to the configured payment gateway. This step verifies that the customer's payment method can cover the total amount without actually charging the card. The authorization typically holds the funds for a short period (usually 7-30 days depending on the payment processor).

7. **Order Record Creation**: With successful payment authorization, the system creates the primary `Order` record containing customer information, shipping address, and order totals. This record is created within the same database transaction that will update inventory levels, ensuring atomicity.

8. **Order Items Creation**: For each product in the checkout session, the system creates corresponding `OrderItem` records. These records capture the product information as it existed at the time of purchase, including name, price, and product snapshot data. This historical preservation is crucial for customer service and financial auditing.
9. **Inventory Deduction**: The system updates the `inventory_quantity` field for each purchased product, subtracting the ordered quantity from available stock. These updates use the previously acquired locks to ensure no conflicts with concurrent transactions.
10. **Payment Capture**: With the order and inventory successfully updated, the system captures the previously authorized payment. This step converts the authorization hold into an actual charge against the customer's payment method.
11. **Cart Session Cleanup**: The system marks the original cart session as converted and clears its contents. This prevents the customer from accidentally purchasing the same items again and frees up session storage space.
12. **Order Confirmation Generation**: Finally, the system generates an order confirmation containing the order number, item details, shipping information, and tracking details. This confirmation is returned to the customer and typically triggers email notifications.

Transaction Rollback Scenarios

The order processing algorithm implements comprehensive rollback procedures to handle failures at any step:

Failure Point	Rollback Actions	Customer Communication
Inventory Check Failure	Release checkout session, no payment processing	"Some items are no longer available. Please review your cart."
Payment Authorization Failure	Release inventory locks, mark session expired	"Payment could not be processed. Please check your payment information."
Order Creation Failure	Void payment authorization, release inventory locks	"An error occurred while creating your order. You have not been charged."
Inventory Update Failure	Void payment, delete order record, release locks	"Order could not be completed due to inventory conflict. You have not been charged."
Payment Capture Failure	Reverse inventory updates, mark order as payment pending	"Order created but payment processing delayed. We will contact you shortly."

Checkout Architecture Decisions

The checkout component's architecture reflects several critical decisions that balance consistency requirements, performance considerations, and integration complexity.

Decision: Pessimistic Locking for Inventory Management

- **Context**: Multiple customers may attempt to purchase the same products simultaneously, particularly during sales events or when inventory levels are low. We need to prevent overselling while maintaining good user experience.
- **Options Considered**: Optimistic locking with conflict resolution, pessimistic locking with row-level locks, eventual consistency with compensation
- **Decision**: Implement pessimistic locking using database row-level locks (SELECT FOR UPDATE) during the checkout process
- **Rationale**: Pessimistic locking provides immediate consistency guarantees and prevents the complex error handling required for optimistic concurrency. While it may reduce throughput during high contention scenarios, it eliminates the possibility of overselling inventory, which is a critical business requirement for e-commerce systems.
- **Consequences**: Enables strong inventory consistency but may create bottlenecks during flash sales. Requires careful timeout management to prevent deadlocks.

Locking Strategy	Pros	Cons	Chosen?
Pessimistic Locking	Immediate consistency, no overselling, simple error handling	Potential bottlenecks, deadlock risk, reduced concurrency	✅ Yes
Optimistic Locking	High concurrency, better performance	Complex conflict resolution, possible overselling, retry logic needed	❌ No
Eventual Consistency	Maximum performance, horizontal scaling	Business complexity, compensation workflows, customer confusion	❌ No

Decision: Synchronous Payment Processing

- **Context**: Payment processing can be handled synchronously during checkout or asynchronously after order creation. Each approach has different implications for user experience and system complexity.
- **Options Considered**: Synchronous payment with order creation, asynchronous payment with order pending state, hybrid approach with authorization and delayed capture
- **Decision**: Use synchronous payment authorization with immediate capture during the checkout transaction
- **Rationale**: Synchronous payment processing provides immediate confirmation to customers and eliminates the complexity of handling pending payment states. While it increases checkout latency, it significantly simplifies the order lifecycle and reduces customer confusion about order status.
- **Consequences**: Enables immediate order confirmation but increases checkout time. Payment gateway timeouts can affect overall system availability.

Payment Timing	Pros	Cons	Chosen?
Synchronous Processing	Immediate confirmation, simple order states, clear customer experience	Higher latency, gateway dependency, timeout risk	✅ Yes
Asynchronous Processing	Lower checkout latency, gateway resilience	Complex order states, customer confusion, reconciliation needed	❌ No
Hybrid Authorization/Capture	Balanced latency, payment flexibility	Additional complexity, auth expiration handling	❌ No

Decision: Checkout Session Isolation

- **Context**: The checkout process spans multiple user interactions and API calls. We need to maintain state consistency across these interactions while preventing interference from cart modifications.
- **Options Considered**: Direct cart modification during checkout, checkout session with cart snapshot, temporary order creation with commit/rollback
- **Decision**: Create isolated checkout sessions that capture cart snapshots and prevent modification of the original cart during checkout
- **Rationale**: Checkout session isolation ensures that the items and prices being processed remain constant throughout the checkout flow, preventing confusion from mid-checkout cart changes. This approach also allows users to abandon checkout and return to cart modification without losing their progress.
- **Consequences**: Requires additional storage for checkout sessions but provides clear state separation and better user experience.

Session Strategy	Pros	Cons	Chosen?
Direct Cart Checkout	Simple implementation, no additional storage	Race conditions, price changes, user confusion	❌ No
Snapshot Isolation	Consistent pricing, clear separation, abandoned checkout handling	Additional storage, session management complexity	✅ Yes
Temporary Orders	Full order flexibility, standard order operations	Complex commit/rollback, order ID confusion	❌ No

Integration Architecture

The checkout component integrates with multiple external services and internal components through well-defined interfaces:

Integration Point	Interface Type	Purpose	Failure Handling
Payment Gateway	HTTP REST API	Payment authorization and capture	Retry with exponential backoff, fallback to manual processing
Address Validation	HTTP REST API	Address verification and standardization	Graceful degradation, allow manual override
Tax Calculation	Internal Service	Calculate sales tax based on jurisdiction	Use cached rates, default to estimated tax
Inventory System	Database Transaction	Atomic inventory updates	Transaction rollback, pessimistic locking
Email Notifications	Message Queue	Order confirmation and updates	Asynchronous processing, retry on failure

Common Checkout Pitfalls

The checkout process involves complex state management and external integrations that create numerous opportunities for subtle bugs and race conditions. Understanding these common pitfalls helps developers implement robust checkout systems that handle edge cases gracefully.

⚠️ Pitfall: Race Conditions on Inventory Updates

The most critical pitfall in checkout systems occurs when multiple customers attempt to purchase the same product simultaneously, particularly when inventory levels are low. Without proper concurrency control, the system may allow overselling, creating customer dissatisfaction and fulfillment problems.

This typically happens when developers implement inventory checks using separate SELECT and UPDATE queries without proper locking. The sequence "check inventory → process payment → update inventory" creates a window where another transaction can modify inventory levels between the check and update operations.

Detection: Monitor for negative inventory levels in the database and customer complaints about orders being canceled after payment processing.

Prevention: Implement pessimistic locking using SELECT FOR UPDATE queries within the same transaction that creates the order. Ensure that inventory checks, payment processing, and inventory updates all occur within a single database transaction or use distributed transaction patterns for multi-database scenarios.

⚠️ Pitfall: Price Staleness Between Cart and Checkout

Users may add items to their cart and then complete checkout hours or days later, during which time product prices may have changed due to sales, promotions, or inventory adjustments. Processing the order with stale prices can result in revenue loss or customer disputes.

This occurs when the checkout process uses price information stored in the cart rather than querying current product prices at checkout time. The problem is exacerbated when cart sessions persist for extended periods without price validation.

Detection: Look for discrepancies between order totals and current product pricing, customer complaints about unexpected charges, or revenue variance reports showing pricing inconsistencies.

Prevention: Always validate current product prices during checkout initialization and re-validate before payment processing. Display clear notifications to customers when prices have changed and require explicit acknowledgment before proceeding.

⚠️ Pitfall: Partial Order Creation During Payment Failures

Payment processing failures can occur after order records are created but before payment is captured, leaving the system in an inconsistent state with unpaid orders and incorrectly allocated inventory.

This happens when order creation and payment processing are not properly coordinated within atomic transactions. Developers often create the order record first and then attempt payment processing, but fail to implement proper cleanup when payment fails.

Detection: Monitor for orders in "pending payment" status that never resolve, inventory discrepancies where stock is allocated to unpaid orders, and customer reports of failed payments with order confirmations.

Prevention: Use database transactions to ensure that order creation, inventory updates, and payment processing either all succeed or all fail together. Implement timeout mechanisms for payment processing and automatic cleanup of failed orders.

⚠️ Pitfall: Session Expiration During Checkout

Users may begin the checkout process but then navigate away or get distracted, returning later to find their session expired and their cart contents potentially lost. This creates frustration and abandoned sales.

The problem occurs when session management doesn't account for the multi-step nature of checkout, applying the same expiration rules to active checkout sessions as to idle browsing sessions.

Detection: Analyze conversion funnel metrics showing high drop-off rates during checkout, customer service reports about lost carts during checkout, and monitoring checkout session expiration patterns.

Prevention: Implement extended session timeouts during active checkout processes and provide clear warnings before session expiration. Implement session recovery mechanisms that can restore checkout state from persistent storage.

⚠️ Pitfall: Address Validation Blocking Valid Addresses

Overly strict address validation can prevent customers from completing orders with legitimate but non-standard addresses, particularly in international markets or rural areas.

This occurs when developers implement rigid address validation rules based on specific postal formats without accounting for regional variations or address standardization differences.

Detection: Monitor checkout abandonment rates at the address collection step, customer complaints about address rejection, and geographic patterns in failed checkouts.

Prevention: Implement progressive address validation that starts with basic format checking and escalates to full validation. Provide manual override options for addresses that fail automatic validation and collect additional verification information when needed.

⚠️ Pitfall: Payment Gateway Timeout Handling

Payment gateway API calls can timeout or fail intermittently, but improper timeout handling can result in duplicate charges or lost orders when customers retry the checkout process.

This happens when developers don't implement proper idempotency controls for payment operations or fail to handle the uncertain state that occurs when payment requests timeout.

Detection: Monitor for duplicate charges, customer complaints about multiple charges for single orders, and payment gateway error logs showing timeout patterns.

Prevention: Implement idempotency keys for all payment operations, provide clear loading states during payment processing, and implement proper retry logic with exponential backoff for transient failures.

⚠️ Pitfall: Insufficient Order Audit Trail

Orders may require modification or cancellation after creation, but insufficient audit logging makes it difficult to track changes and maintain financial accuracy.

This occurs when developers focus only on the successful order creation path without implementing comprehensive logging for order state changes, payment adjustments, or administrative modifications.

Detection: Difficulty reconciling payment processor records with internal order data, challenges in customer service investigations, and regulatory compliance issues during audits.

Prevention: Implement comprehensive audit logging for all order state changes, payment events, and administrative actions. Store immutable order snapshots and maintain detailed change logs with timestamps and user attribution.

Implementation Guidance

The checkout process implementation requires careful coordination of multiple system components while maintaining transactional integrity and providing clear error handling. This section provides specific technical guidance for implementing a robust checkout system.

Technology Recommendations

Component	Simple Option	Advanced Option
Database Transactions	SQLite with WAL mode and explicit transactions	PostgreSQL with row-level locking and serializable isolation
Payment Processing	Stripe API with webhook handling	Multiple payment gateways with adapter pattern
Address Validation	Basic regex validation with manual override	Google Maps Geocoding API with SmartyStreets fallback
Session Management	Express-session with database store	Redis-based sessions with clustering support
Error Monitoring	Console logging with structured format	Sentry error tracking with performance monitoring

Recommended File Structure

```
src/
  components/
    checkout/
      CheckoutController.js     ← API route handlers
      CheckoutService.js        ← Business logic and transaction coordination
      OrderService.js           ← Order creation and management
      PaymentService.js         ← Payment gateway integration
      AddressValidator.js       ← Address validation and standardization
      CheckoutModels.js         ← Data models and validation schemas
      checkout.test.js          ← Integration tests for checkout flow
  middleware/
    transactionMiddleware.js    ← Database transaction management
    checkoutAuth.js            ← Checkout session authentication
  config/
    payment.js                 ← Payment gateway configuration
  database/
    migrations/
      004_create_checkout_tables.sql  ← Database schema for checkout
```

Core Database Schema

```sql
-- Checkout sessions for multi-step process isolation
CREATE TABLE checkout_sessions (
    id TEXT PRIMARY KEY,
    session_id TEXT NOT NULL,
    user_id INTEGER,
    status TEXT NOT NULL CHECK (status IN ('started', 'address_collected', 'validated', 'processing',
'completed', 'expired')),
    cart_snapshot TEXT NOT NULL,  -- JSON snapshot of cart contents
    shipping_address TEXT,        -- JSON address information
    billing_address TEXT,         -- JSON billing address
    price_total INTEGER,          -- Total in cents
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    expires_at DATETIME NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

-- Orders table for completed purchases
CREATE TABLE orders (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER,
    session_id TEXT,
    order_number TEXT UNIQUE NOT NULL,
    status TEXT NOT NULL CHECK (status IN ('pending', 'confirmed', 'processing', 'shipped', 'delivered',
'cancelled')),
    shipping_address TEXT NOT NULL,  -- JSON address
    billing_address TEXT NOT NULL,   -- JSON address
    subtotal INTEGER NOT NULL,       -- Amount in cents
    shipping_cost INTEGER NOT NULL,  -- Amount in cents
    tax_amount INTEGER NOT NULL,     -- Amount in cents
    total_amount INTEGER NOT NULL,   -- Amount in cents
    payment_method TEXT,             -- Payment gateway reference
    payment_status TEXT NOT NULL CHECK (payment_status IN ('pending', 'authorized', 'captured', 'failed',
'refunded')),
```

```
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Checkout Service Infrastructure

```javascript
// CheckoutService.js - Complete service class for checkout operations

const { v4: uuidv4 } = require('uuid');

const Database = require('../database/Database');

const PaymentService = require('./PaymentService');

const AddressValidator = require('./AddressValidator');


class CheckoutService {

    constructor(database, paymentService, addressValidator) {

        this.db = database;

        this.payment = paymentService;

        this.addressValidator = addressValidator;

        this.CHECKOUT_TIMEOUT_MS = 15 * 60 * 1000; // 15 minutes

    }


    // Initialize checkout session with cart snapshot

    async startCheckout(sessionId) {

        const db = this.db.getConnection();



        return db.serialize(() => {

            db.run('BEGIN TRANSACTION');



            try {

                // Get current cart contents

                const cartItems = db.prepare(`

                    SELECT ci.*, p.name, p.price, p.inventory_quantity

                    FROM cart_items ci

                    JOIN products p ON ci.product_id = p.id

                    WHERE ci.cart_id = (SELECT id FROM carts WHERE session_id = ?)

                    AND p.is_active = 1

                `).all(sessionId);


                if (cartItems.length === 0) {
```

```
                throw new Error('Cart is empty or contains only inactive products');
            }

            // Create checkout session with cart snapshot
            const checkoutId = uuidv4();
            const expiresAt = new Date(Date.now() + this.CHECKOUT_TIMEOUT_MS);

            db.prepare(`
                INSERT INTO checkout_sessions
                (id, session_id, status, cart_snapshot, expires_at)
                VALUES (?, ?, 'started', ?, ?)
            `).run(checkoutId, sessionId, JSON.stringify(cartItems), expiresAt);

            db.run('COMMIT');

            return {
                checkoutId,
                items: cartItems,
                expiresAt,
                status: 'started'
            };

        } catch (error) {
            db.run('ROLLBACK');
            throw error;
        }
    });
}

// Skeleton for address collection with validation
async collectAddress(checkoutId, shippingAddress, billingAddress = null) {
    // TODO 1: Validate checkout session exists and hasn't expired
    // TODO 2: Validate shipping address format using AddressValidator
```

```
        // TODO 3: If no billing address provided, use shipping address
        // TODO 4: Update checkout session with address information
        // TODO 5: Calculate shipping costs based on address and cart contents
        // TODO 6: Update checkout session status to 'address_collected'
        // TODO 7: Return updated checkout session with shipping costs

        throw new Error('collectAddress not implemented');
    }

    // Skeleton for final validation before order creation
    async validateForCheckout(checkoutId) {
        // TODO 1: Get checkout session and verify it's in correct status
        // TODO 2: Parse cart snapshot and validate against current product data
        // TODO 3: Check inventory availability for each item using SELECT FOR UPDATE
        // TODO 4: Validate prices haven't changed significantly (>5% threshold)
        // TODO 5: Calculate final totals including shipping and tax
        // TODO 6: Update checkout session status to 'validated'
        // TODO 7: Return validation report with any issues found

        throw new Error('validateForCheckout not implemented');
    }

    // Skeleton for order creation and payment processing
    async confirmOrder(checkoutId, paymentMethod) {
        const db = this.db.getConnection();

        return db.serialize(() => {
            db.run('BEGIN TRANSACTION');

            try {
                // TODO 1: Get validated checkout session
                // TODO 2: Lock inventory for all products in order (SELECT FOR UPDATE)
                // TODO 3: Authorize payment through payment service
```

```
                // TODO 4: Create order record with generated order number
                // TODO 5: Create order_items records for each product
                // TODO 6: Update product inventory quantities
                // TODO 7: Capture authorized payment
                // TODO 8: Mark checkout session as completed
                // TODO 9: Clear original cart session
                // TODO 10: Generate order confirmation with tracking information

                throw new Error('confirmOrder not implemented');

            } catch (error) {
                db.run('ROLLBACK');
                // TODO: Handle specific error types (inventory, payment, etc.)
                throw error;
            }
        });
    }
}

module.exports = CheckoutService;
```

Payment Service Integration

JAVASCRIPT

```javascript
// PaymentService.js - Payment gateway abstraction

const stripe = require('stripe')(process.env.STRIPE_SECRET_KEY);


class PaymentService {

    // Authorize payment without capturing funds

    async authorizePayment(amount, paymentMethod, metadata = {}) {

        try {

            const paymentIntent = await stripe.paymentIntents.create({

                amount: amount, // Amount in cents

                currency: 'usd',

                payment_method: paymentMethod.id,

                confirmation_method: 'manual',

                confirm: true,

                capture_method: 'manual', // Authorize only, don't capture

                metadata: {

                    order_id: metadata.orderId,

                    customer_email: metadata.customerEmail

                }

            });


            return {

                success: true,

                authorizationId: paymentIntent.id,

                status: paymentIntent.status,

                amount: paymentIntent.amount

            };



        } catch (error) {

            return {

                success: false,

                error: error.message,

                code: error.code
```

```
        };
    }
}

// Capture previously authorized payment
async capturePayment(authorizationId, amount = null) {
    try {
        const paymentIntent = await stripe.paymentIntents.capture(
            authorizationId,
            amount ? { amount_to_capture: amount } : {}
        );

        return {
            success: true,
            captureId: paymentIntent.id,
            status: paymentIntent.status,
            amount: paymentIntent.amount_received
        };

    } catch (error) {
        return {
            success: false,
            error: error.message,
            code: error.code
        };
    }
}

// Void authorization if order fails
async voidAuthorization(authorizationId) {
    // Implementation for voiding payment authorization
    // This prevents funds from being held on customer's card
}
```

```
}

module.exports = PaymentService;
```

Address Validator Implementation

JAVASCRIPT

```javascript
// AddressValidator.js - Address validation with progressive verification

class AddressValidator {

    constructor() {

        this.requiredFields = ['fullName', 'addressLine1', 'city', 'state', 'postalCode', 'country'];

    }


    // Basic format validation

    validateFormat(address) {

        const errors = [];


        // Full name validation

        if (!address.fullName || address.fullName.length < 2 || address.fullName.length > 100) {

            errors.push('Name must be 2-100 characters');

        }


        // Address line validation

        if (!address.addressLine1 || address.addressLine1.length < 5) {

            errors.push('Street address must be at least 5 characters');

        }


        // Postal code validation (basic US format)

        if (!address.postalCode || !/^\d{5}(-\d{4})?$/.test(address.postalCode)) {

            errors.push('Please enter a valid 5 or 9 digit ZIP code');

        }


        return {

            isValid: errors.length === 0,

            errors

        };

    }


    // Enhanced validation with geocoding (stub for external service)

    async validateWithGeocoding(address) {

        const formatValidation = this.validateFormat(address);
```

```
        if (!formatValidation.isValid) {
            return formatValidation;
        }

        // TODO: Integrate with Google Maps Geocoding API
        // For now, return format validation result
        return {
            isValid: true,
            standardizedAddress: address,
            confidence: 'high'
        };
    }
}

module.exports = AddressValidator;
```

Error Handling Patterns

Error Type	HTTP Status	Response Format	Client Action
Cart Empty	400	`{"error": "CART_EMPTY", "message": "Cart contains no items"}`	Redirect to catalog
Session Expired	409	`{"error": "SESSION_EXPIRED", "message": "Checkout session has expired"}`	Restart checkout
Inventory Conflict	409	`{"error": "INVENTORY_UNAVAILABLE", "items": [...]}`	Update cart and retry
Payment Failed	402	`{"error": "PAYMENT_FAILED", "message": "Card declined"}`	Retry with different payment
Address Invalid	400	`{"error": "ADDRESS_INVALID", "field": "postalCode"}`	Correct address and resubmit

Milestone Checkpoint

After implementing the checkout process component, verify functionality through these checkpoints:

Test Checkout Flow:

1. Start with items in cart: `POST /api/checkout/start` should return checkout session
2. Submit valid address: `POST /api/checkout/address` should accept and calculate shipping
3. Get order summary: `GET /api/checkout/summary` should show itemized totals

4. Complete order: `POST /api/checkout/confirm` should create order and process payment

Verify Transaction Safety:

1. Simulate payment failure after order creation - order should not exist in database
2. Attempt checkout with insufficient inventory - should fail gracefully
3. Try concurrent checkouts for same product - only one should succeed

Expected Database State:

- Completed orders should have corresponding `order_items` records
- Product inventory should be decremented by ordered quantities
- Cart session should be cleared after successful checkout
- Checkout session should be marked as completed

Error Handling Verification:

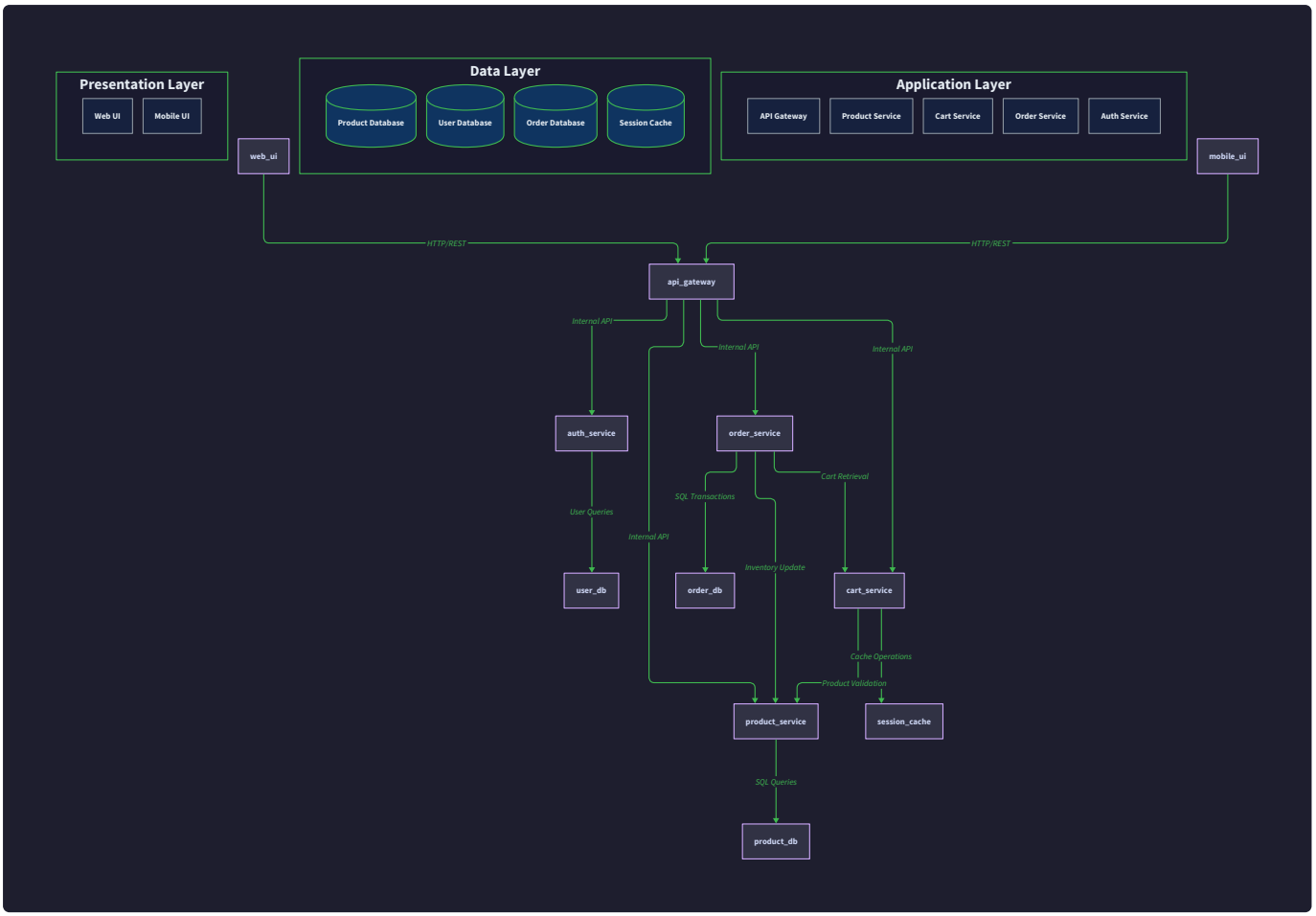
- Invalid addresses should return specific field errors
- Expired sessions should require restart of checkout process
- Payment failures should preserve cart contents for retry

Interactions and Data Flow

> **Milestone(s):** All milestones - describes how product catalog, shopping cart, user authentication, and checkout components communicate and coordinate during key user operations

Understanding how components work together is like watching the choreographed dance behind a successful retail operation. While customers see a simple storefront experience - browsing products, adding items to their cart, and checking out - the underlying system orchestrates dozens of precise interactions between specialized components. Each component has its role, but the magic happens in their coordination: inventory systems preventing overselling, cart managers synchronizing state across page refreshes, authentication services maintaining secure sessions, and checkout processors ensuring atomic order creation.

The interaction patterns in our e-commerce system follow a **request-response architecture** where the frontend client initiates operations by calling REST API endpoints, which then orchestrate calls between internal components. This creates a clear separation between user-facing operations and internal business logic, allowing each component to evolve independently while maintaining consistent external interfaces.



Component Communication

The communication between components follows a **layered interaction model** where each tier communicates only with adjacent layers, preventing the tight coupling that would make the system brittle and hard to maintain. Think of this like a restaurant's hierarchy: customers don't directly communicate with the kitchen staff; they tell the waiter, who communicates with the kitchen, which coordinates with the inventory manager and the billing system.

API Layer Communication

The API layer serves as the primary orchestration point, receiving HTTP requests from the frontend and coordinating between business logic components. Each API endpoint follows a consistent pattern of validation, business logic delegation, and response formatting. The API layer never directly manipulates data - instead, it delegates all business operations to specialized components and focuses solely on HTTP protocol concerns.

Communication Pattern	Source	Target	Interface Type	Data Format
Frontend to API	Client Browser	Express Router	HTTP REST	JSON payloads
API to Business Logic	Route Handlers	Component Classes	Function Calls	JavaScript Objects
Business Logic to Data	Component Methods	Database Layer	SQL Queries	Prepared Statements
Component to Component	Product Catalog	Cart Manager	Direct Method Calls	Structured Objects
Session Management	Authentication	All Components	Middleware Pipeline	Session Objects

Inter-Component Interface Contracts

Each component exposes well-defined interfaces that hide internal implementation details while providing stable contracts for communication. These interfaces use **dependency injection patterns** where components receive references to other components they need, rather than creating those dependencies directly.

Interface	Providing Component	Consuming Component	Contract Type	Error Handling
Product Query Interface	Product Catalog	Cart Manager	Synchronous	Exception Propagation
Inventory Interface	Product Catalog	Checkout Process	Synchronous	Validation Results
Session Interface	Authentication	All Components	Middleware	Session Validation
Cart Interface	Cart Manager	Checkout Process	Synchronous	State Objects
Order Interface	Checkout Process	Order Management	Transactional	Atomic Operations

The **Product Query Interface** provides the primary mechanism for other components to retrieve product information without directly accessing the database. This interface supports both individual product lookups and batch operations for cart validation scenarios:

Method	Parameters	Returns	Purpose	Error Conditions
`findById(productId)`	productId: number	Product or null	Single product retrieval	Product not found, invalid ID
`findByIds(productIds)`	productIds: number[]	Product[]	Batch product retrieval	Some products not found
`checkInventory(productId, quantity)`	productId: number, quantity: number	InventoryResult	Availability validation	Insufficient stock
`reserveInventory(items)`	items: CartItem[]	ReservationResult	Temporary inventory hold	Reservation conflicts

Session-Based State Coordination

Session management creates the coordination backbone that allows stateless HTTP requests to maintain consistent user state across the shopping journey. The session system acts like a theater's coat check - providing a claim ticket (session ID) that can retrieve a customer's accumulated state (cart contents, authentication status, checkout progress) from any interaction point.

The session coordination follows a **middleware pattern** where authentication status and cart state are resolved early in each request lifecycle, making this information available to all subsequent business logic without additional lookups:

Session Scope	State Maintained	Coordination Method	Persistence Strategy
Authentication	User identity and permissions	Session middleware	Database session store
Cart State	Items, quantities, pricing	Session-linked cart ID	Database with TTL cleanup
Checkout Progress	Address, payment method	Checkout session ID	Time-limited database records
Anonymous Browsing	Product views, search history	Client-side storage	Browser localStorage

> **Design Insight**: Session-based coordination eliminates the need for components to directly communicate about user state. Instead of the cart component asking the authentication component "who is this user?", the middleware resolves identity once per request and provides it as context to all components.

Component Dependency Architecture

The component dependency structure follows a **directed acyclic graph** where higher-level components depend on lower-level ones, but dependencies never form cycles. This ensures clean separation of concerns and prevents the circular dependencies that would make testing and maintenance difficult.

```
API Layer
├── Authentication Component (session management)
├── Product Catalog Component (product queries)
├── Cart Manager Component
│   └── depends on: Product Catalog (inventory validation)
│   └── depends on: Authentication (user context)
└── Checkout Processor Component
    └── depends on: Cart Manager (cart contents)
    └── depends on: Product Catalog (final inventory check)
    └── depends on: Authentication (user verification)
```

This dependency structure means that components at the same level never directly communicate - they only interact through their common dependencies or through the API layer orchestration. For example, the Authentication Component never directly calls Cart Manager methods; instead, authentication middleware provides user context that Cart Manager consumes when called by API handlers.

User Journey Flows

The user journey flows represent the end-to-end sequences of component interactions that deliver complete customer experiences. Each journey maps to specific business scenarios and involves precise coordination between multiple components, with each step building on the results of previous steps.

Product Discovery Journey

The product discovery journey begins when a customer wants to find products, whether through browsing categories, searching by keywords, or filtering by attributes. This journey showcases how the Product Catalog Component coordinates with the API layer to deliver paginated, filtered, and sorted product information.

Consider a customer searching for "wireless headphones" under $100:

1. **Client Request Initiation**: Browser sends GET request to `/api/products/search?q=wireless+headphones&maxPrice=100&page=1&limit=20&sort=price_asc`
2. **API Request Parsing**: Express route handler validates query parameters, checking that price values are numbers, page/limit are positive integers, and sort options are from allowed values
3. **Search Query Construction**: API handler constructs search criteria object: `{ searchTerms: ['wireless', 'headphones'], priceRange: { max: 10000 }, pagination: { page: 1, limit: 20 }, sorting: { field: 'price', direction: 'asc' } }`
4. **Catalog Component Invocation**: Route handler calls `productCatalog.searchProducts(searchCriteria)` with the constructed criteria object
5. **Database Query Execution**: Product Catalog Component builds SQL query with full-text search on name/description fields, price filtering with WHERE clause, and ORDER BY with LIMIT/OFFSET for pagination
6. **Result Set Processing**: Component processes raw database rows, calculating pagination metadata (total count, page count, has next/previous flags), and formatting product objects with computed fields like discounted prices
7. **Response Assembly**: API handler receives `ProductListResponse` object and formats HTTP response with products array, pagination metadata, and search result statistics

8. **Client State Update**: Frontend receives response and updates product listing UI, search filters, and pagination controls

The key interaction pattern here is the **query delegation model** where the API layer focuses on HTTP protocol concerns while the Product Catalog Component handles all business logic around search, filtering, and pagination.

Cart Management Journey

The cart management journey demonstrates **session-based state coordination** as customers add, update, and remove items while maintaining consistency across page refreshes and browser sessions. This journey involves the most complex component interactions because cart operations must validate inventory, maintain pricing accuracy, and handle both anonymous and authenticated sessions.

Consider a customer adding a product to their cart, then updating the quantity:

1. **Add to Cart Initiation**: Browser sends POST request to `/api/cart/items` with `{ productId: 123, quantity: 2 }` payload
2. **Session Resolution**: Authentication middleware extracts session ID from HTTP cookie, determines if user is authenticated or anonymous, and provides session context to subsequent handlers
3. **Product Validation**: Cart Manager calls `productCatalog.findById(123)` to verify product exists, is active, and has sufficient inventory
4. **Inventory Check**: Product Catalog Component queries current inventory level and compares against requested quantity, returning validation result
5. **Cart State Retrieval**: Cart Manager queries database for existing cart linked to session ID, creating new cart record if none exists
6. **Item Addition Logic**: Cart Manager checks if product already exists in cart - if yes, increments quantity; if no, creates new CartItem record with current product price snapshot
7. **Cart Totals Calculation**: Cart Manager recalculates cart subtotal, item count, and identifies any price changes since items were originally added
8. **Persistent State Update**: Database transaction commits cart changes atomically, ensuring inventory checks and cart updates happen together
9. **Response Generation**: Cart Manager returns `CartSummary` object with updated items, totals, and any price change notifications
10. **Quantity Update Request**: Customer later sends PUT request to `/api/cart/items/456` with `{ quantity: 3 }`
11. **Quantity Validation**: Cart Manager validates new quantity against current inventory, preventing updates that would exceed available stock
12. **Optimistic Update Pattern**: Cart Manager updates quantity immediately but flags the change for inventory revalidation during checkout

The cart journey showcases **eventual consistency patterns** where the system optimistically allows cart updates but performs comprehensive validation at checkout boundaries to ensure inventory accuracy.

Authentication Journey

The authentication journey establishes the security context that enables personalized experiences and order processing. This journey demonstrates **stateful session management** and the coordination between password security, session creation, and cart state transfer.

Consider a customer registering a new account during their shopping session:

1. **Registration Form Submission**: Browser sends POST request to `/api/auth/register` with `{ email: 'customer@example.com', password: 'SecurePass123!', confirmPassword: 'SecurePass123!' }`
2. **Input Validation Pipeline**: Authentication Component validates email format using regex, confirms password meets strength requirements (length, complexity), and verifies password confirmation matches
3. **Duplicate Account Check**: Component queries user database to ensure email address isn't already registered, returning validation error if duplicate found
4. **Password Security Processing**: Component generates cryptographic salt and uses `bcrypt` with `SALT_ROUNDS` configuration to create password hash, never storing plaintext password
5. **User Account Creation**: Database transaction creates new User record with email, password hash, and timestamp fields, generating auto-increment user ID
6. **Session Establishment**: Authentication Component calls `createSession(userId)` to generate cryptographically secure session ID and store session record with expiration timestamp
7. **Cookie Configuration**: API handler sets HTTP-only session cookie with secure flags, preventing JavaScript access and ensuring HTTPS-only transmission
8. **Anonymous Cart Transfer**: If customer had items in anonymous cart, Cart Manager transfers cart ownership from anonymous session ID to authenticated user ID
9. **Welcome Response**: API returns success response with user profile information and updated cart contents reflecting any merged items

The authentication journey illustrates **security layering** where multiple validation steps and cryptographic protections work together to establish trusted user identity.

Checkout Process Journey

The checkout journey represents the most complex component coordination scenario, involving **transactional operations** that must maintain consistency across inventory updates, cart conversion, order creation, and payment processing. This journey showcases **pessimistic locking patterns** and **atomic state transitions**.

Consider a customer completing their purchase:

1. **Checkout Initiation**: Browser sends POST request to `/api/checkout/start` to begin checkout process
2. **Cart Snapshot Creation**: Checkout Processor calls `cartManager.getCartContents(sessionId)` to capture current cart state, creating immutable checkout session with fixed pricing and inventory requirements
3. **Address Collection**: Customer submits shipping address through PUT request to `/api/checkout/address`, triggering address format validation and optional address verification service calls
4. **Final Inventory Validation**: Checkout Processor begins database transaction with `SELECT FOR UPDATE` queries on product inventory to prevent concurrent modifications during checkout completion
5. **Payment Authorization**: Payment gateway integration authorizes payment method for full order amount without capturing funds, ensuring payment method is valid and has sufficient balance
6. **Order Creation Transaction**: Within single database transaction:
    - Creates Order record with generated order number
    - Creates OrderItem records for each cart item with quantity and price snapshots
    - Decrements product inventory by ordered quantities
    - Marks cart as converted and checkout session as completed

7. **Payment Capture**: After successful order creation, payment authorization is captured to complete financial transaction
8. **Confirmation Generation**: System generates order confirmation with order number, estimated delivery date, and itemized receipt
9. **State Cleanup**: Cart is cleared, checkout session is marked completed, and any temporary inventory reservations are released

The checkout journey demonstrates **ACID transaction principles** where all operations must succeed together or fail together, preventing partial orders or inventory inconsistencies.

> **Critical Design Principle**: Each user journey follows the **Command Query Responsibility Segregation (CQRS)** pattern where read operations (browsing products, viewing cart) are optimized for performance and user experience, while write operations (adding to cart, placing orders) prioritize consistency and data integrity.

API Contracts

The API contracts define the precise interfaces between the frontend client and backend services, specifying request formats, response structures, status codes, and error conditions. These contracts serve as the formal specification that enables frontend and backend development to proceed independently while ensuring compatibility.

Product Catalog API Contract

The Product Catalog API provides endpoints for product discovery, search, and detailed product information. These endpoints support both human browsing experiences and programmatic integrations for inventory management.

Endpoint	Method	Parameters	Request Body	Success Response	Error Responses
`/api/products`	GET	`page`, `limit`, `sort`, `category`, `minPrice`, `maxPrice`	None	200 with `ProductListResponse`	400 for invalid parameters
`/api/products/search`	GET	`q`, `page`, `limit`, `sort`, `category`	None	200 with `ProductListResponse`	400 for invalid query
`/api/products/:id`	GET	`id` (path parameter)	None	200 with `ProductDetailResponse`	404 if product not found
`/api/categories`	GET	`parent` (optional)	None	200 with `CategoryListResponse`	500 for database errors
`/api/search/suggestions`	GET	`q`, `limit`	None	200 with `SearchSuggestionResponse`	400 for empty query

Product List Response Structure

The `ProductListResponse` follows a standard pagination envelope pattern that provides both data and metadata necessary for client-side pagination and filtering UI:

Field	Type	Description	Example Value
`products`	Product[]	Array of product objects matching query criteria	`[{id: 1, name: "Laptop"...}]`
`pagination`	Object	Pagination metadata with current page and totals	`{page: 1, limit: 20, total: 150}`
`totalPages`	number	Total number of pages available	`8`
`hasNextPage`	boolean	Whether additional pages exist	`true`
`hasPrevPage`	boolean	Whether previous pages exist	`false`
`appliedFilters`	Object	Summary of active search and filter criteria	`{category: "Electronics", priceRange: {...}}`
`sortOptions`	Object	Available sort criteria and current selection	`{current: "price_asc", available: [...]}`

Product Detail Response Structure

Individual product responses include comprehensive information needed for product detail pages and cart operations:

Field	Type	Description	Validation Rules
`id`	number	Unique product identifier	Positive integer, must exist
`name`	string	Product display name	1-200 characters, required
`description`	string	Detailed product description	Up to 2000 characters
`price`	number	Current price in cents	Positive integer, required
`originalPrice`	number	Original price before discounts	Positive integer, may be null
`inventoryQuantity`	number	Available stock level	Non-negative integer
`category`	Object	Category information with hierarchy	`{id, name, path}`
`images`	string[]	Array of image URLs	Valid URLs, at least one required
`attributes`	Object	Product-specific attributes	Key-value pairs, product-dependent
`isActive`	boolean	Whether product is available for purchase	Must be true for cart operations

Shopping Cart API Contract

The Shopping Cart API manages cart state with operations for adding, updating, and removing items. All cart operations require session identification through HTTP cookies and return updated cart summaries.

Endpoint	Method	Parameters	Request Body	Success Response	Error Responses
`/api/cart`	GET	None	None	200 with `CartSummary`	401 if session invalid
`/api/cart/items`	POST	None	`{productId, quantity}`	201 with `CartSummary`	400 for invalid data, 409 for insufficient inventory
`/api/cart/items/:id`	PUT	`id` (cart item ID)	`{quantity}`	200 with `CartSummary`	404 if item not in cart, 409 for inventory issues
`/api/cart/items/:id`	DELETE	`id` (cart item ID)	None	200 with `CartSummary`	404 if item not in cart
`/api/cart/validate`	POST	None	None	200 with `ValidationReport`	500 for validation errors

Cart Summary Response Structure

The `CartSummary` provides complete cart state information and pricing calculations:

Field	Type	Description	Calculation Method
`items`	CartItem[]	Array of items currently in cart	Direct from database with product joins
`itemCount`	number	Total number of individual items	Sum of all item quantities
`subtotal`	number	Total price before taxes and shipping	Sum of (quantity × price_snapshot)
`estimatedTax`	number	Estimated tax amount	Calculated based on shipping address
`estimatedTotal`	number	Subtotal plus estimated tax	`subtotal + estimatedTax`
`priceChanges`	Object[]	Items with price changes since addition	Comparison of current vs snapshot prices
`inventoryIssues`	Object[]	Items with insufficient inventory	Current availability vs requested quantity
`lastUpdated`	string	ISO timestamp of last cart modification	Updated on every cart operation

Add to Cart Request Structure

Cart addition requests require product identification and quantity specification:

Field	Type	Required	Validation Rules	Error Messages
`productId`	number	Yes	Must be positive integer, product must exist	"Product not found"
`quantity`	number	Yes	Must be positive integer, ≤ available inventory	"Insufficient inventory"
`priceSnapshot`	number	No	If provided, must match current price	"Price has changed"

User Authentication API Contract

The Authentication API handles user registration, login, logout, and session management with comprehensive security validation and error handling.

Endpoint	Method	Parameters	Request Body	Success Response	Error Responses
`/api/auth/register`	POST	None	`{email, password, confirmPassword}`	201 with user profile	400 for validation errors, 409 for duplicate email
`/api/auth/login`	POST	None	`{email, password}`	200 with user profile	400 for invalid format, 401 for wrong credentials
`/api/auth/logout`	POST	None	None	200 with success message	401 if not authenticated
`/api/auth/profile`	GET	None	None	200 with user profile	401 if session invalid
`/api/auth/change-password`	PUT	None	`{currentPassword, newPassword}`	200 with success message	400 for validation, 401 for wrong current password

Registration Request Structure

User registration requires email and password with confirmation:

Field	Type	Required	Validation Rules	Error Messages
`email`	string	Yes	Valid email format, not already registered	"Invalid email format", "Email already exists"
`password`	string	Yes	≥8 characters, mixed case, numbers, symbols	"Password too weak"
`confirmPassword`	string	Yes	Must exactly match password field	"Passwords do not match"

Authentication Response Structure

Successful authentication returns user profile information and session establishment:

Field	Type	Description	Privacy Notes
`userId`	number	Unique user identifier	Safe to expose to authenticated user
`email`	string	User's email address	Only returned to the user themselves
`createdAt`	string	Account creation timestamp	ISO 8601 format
`sessionExpiresAt`	string	Session expiration timestamp	For client-side session management

Checkout Process API Contract

The Checkout API orchestrates the multi-step process from cart review to order completion, with each endpoint building on previous steps in the checkout flow.

Endpoint	Method	Parameters	Request Body	Success Response	Error Responses
`/api/checkout/start`	POST	None	None	201 with `CheckoutSession`	400 if cart empty, 401 if not authenticated
`/api/checkout/:id/address`	PUT	`id` (checkout ID)	`{shippingAddress, billingAddress}`	200 with `CheckoutSession`	400 for invalid address, 404 for invalid checkout ID
`/api/checkout/:id/validate`	POST	`id` (checkout ID)	None	200 with `ValidationReport`	409 for inventory issues
`/api/checkout/:id/confirm`	POST	`id` (checkout ID)	`{paymentMethod}`	201 with `OrderConfirmation`	400 for payment issues, 409 for validation failures

Address Structure

Address information requires comprehensive validation for shipping and billing:

Field	Type	Required	Validation Rules	Format Requirements
`fullName`	string	Yes	2-100 characters	No special characters except hyphens, apostrophes
`addressLine1`	string	Yes	5-100 characters	Street address with number and name
`addressLine2`	string	No	Up to 100 characters	Apartment, suite, unit information
`city`	string	Yes	2-50 characters	Valid city name
`state`	string	Yes	2 character code	Valid state/province abbreviation
`postalCode`	string	Yes	5-10 characters	Format depends on country
`country`	string	Yes	2 character code	ISO 3166-1 alpha-2 country code
`phoneNumber`	string	Yes	10-15 digits	Valid phone format for delivery coordination

Order Confirmation Response Structure

Order confirmation provides comprehensive receipt information and next steps:

Field	Type	Description	Purpose
`orderId`	number	Internal order identifier	Database primary key
`orderNumber`	string	Customer-facing order reference	Formatted for customer service
`status`	string	Current order status	"confirmed", "processing", "shipped"
`items`	OrderItem[]	Itemized list of purchased products	Receipt detail
`pricing`	Object	Breakdown of subtotal, tax, shipping, total	Financial record
`shippingAddress`	Address	Delivery address	Shipping coordination
`estimatedDelivery`	string	Expected delivery date range	Customer expectation setting
`trackingInfo`	Object	Shipping tracking details	Package monitoring

> **API Design Principle**: All API responses follow the **envelope pattern** where data is wrapped in metadata objects that provide context, pagination, validation results, and operational information. This creates consistent client-side handling patterns and enables rich user interface experiences.

Status Code Conventions

The API follows REST conventions for HTTP status codes with specific meanings for e-commerce operations:

Status Code	Usage Context	Meaning	Example Scenarios
200 OK	Successful read operations	Request completed successfully	Get cart contents, view product details
201 Created	Successful resource creation	New resource created	User registration, order placement
400 Bad Request	Client input validation failures	Request format or content invalid	Invalid email format, negative quantities
401 Unauthorized	Authentication failures	Request requires valid session	Accessing cart without login
403 Forbidden	Authorization failures	Valid session but insufficient permissions	Admin operations from regular user
404 Not Found	Resource lookup failures	Requested resource doesn't exist	Invalid product ID, nonexistent cart item
409 Conflict	Business rule violations	Request conflicts with current state	Insufficient inventory, duplicate email
422 Unprocessable Entity	Semantic validation failures	Request format valid but content invalid	Checkout with empty cart
500 Internal Server Error	System failures	Unexpected server-side error	Database connection failures

Error Response Structure

All error responses follow a consistent format that enables client-side error handling and user-friendly error messages:

Field	Type	Always Present	Description	Example Value
`error`	string	Yes	High-level error category	"VALIDATION_FAILED"
`message`	string	Yes	Human-readable error description	"The requested quantity exceeds available inventory"
`details`	Object[]	No	Field-specific validation errors	`[{field: "quantity", message: "Must be positive"}]`
`code`	string	Yes	Machine-readable error identifier	"INSUFFICIENT_INVENTORY"
`requestId`	string	Yes	Unique request identifier for debugging	"req_1234567890"
`timestamp`	string	Yes	Error occurrence timestamp	"2024-01-15T10:30:00Z"

Implementation Guidance

This section provides concrete implementation patterns and starter code for building the interaction layer that coordinates between components and manages data flow in the e-commerce system.

Technology Recommendations

Component	Simple Option	Advanced Option	Recommended For
HTTP Server	Express.js with built-in JSON parsing	Fastify with custom serialization	Express for learning simplicity
Session Management	express-session with memory store	Redis-backed sessions with clustering	Memory store for development
Request Validation	Manual validation with joi or yup	JSON Schema with ajv validator	joi for readable validation rules
API Documentation	Manual documentation with examples	OpenAPI/Swagger with auto-generation	Manual docs for flexibility
Error Handling	Custom error classes with status codes	Error tracking with Sentry integration	Custom classes for learning
Response Formatting	Consistent response wrapper functions	Serialization middleware pipeline	Wrapper functions for simplicity

Project File Structure

The interaction layer organizes around API routes and coordination logic:

```
src/
├── api/
│   ├── routes/
│   │   ├── products.js        ← Product catalog endpoints
│   │   ├── cart.js            ← Shopping cart endpoints
│   │   ├── auth.js            ← Authentication endpoints
│   │   └── checkout.js        ← Checkout process endpoints
│   ├── middleware/
│   │   ├── auth.js            ← Session validation middleware
│   │   ├── validation.js      ← Request validation middleware
│   │   └── error-handler.js   ← Centralized error handling
│   └── schemas/
│       ├── product.js         ← Request/response validation schemas
│       ├── cart.js            ← Cart operation schemas
│       └── checkout.js        ← Checkout flow schemas
├── components/
│   ├── ProductCatalog.js      ← Business logic components
│   ├── CartManager.js         ← (implemented in previous sections)
│   ├── Authentication.js
│   └── CheckoutProcessor.js
└── app.js                     ← Main application setup
```

Request Coordination Infrastructure

This complete middleware setup handles session management, validation, and error handling:

JAVASCRIPT

```javascript
// src/api/middleware/auth.js

const jwt = require('jsonwebtoken');

const { Session } = require('../../models');


class AuthenticationMiddleware {

  static async validateSession(req, res, next) {

    try {

      const sessionId = req.cookies.sessionId;



      if (!sessionId) {

        req.session = { isAuthenticated: false, sessionId: null };

        return next();

      }


      const session = await Session.findBySessionId(sessionId);

      if (!session || session.isExpired()) {

        res.clearCookie('sessionId');

        req.session = { isAuthenticated: false, sessionId: null };

        return next();

      }


      // Update last activity

      await session.updateActivity();



      req.session = {

        isAuthenticated: true,

        sessionId: session.id,

        userId: session.userId,

        user: await session.getUser()

      };



      next();

    } catch (error) {
```

```
      next(new Error('Session validation failed'));
    }
  }

  static requireAuthentication(req, res, next) {
    if (!req.session.isAuthenticated) {
      return res.status(401).json({
        error: 'AUTHENTICATION_REQUIRED',
        message: 'This operation requires authentication',
        code: 'AUTH_REQUIRED'
      });
    }
    next();
  }
}

module.exports = AuthenticationMiddleware;
```

Response Formatting Infrastructure

Standardized response formatting ensures consistent API contracts:

JAVASCRIPT

```javascript
// src/api/middleware/response-formatter.js

class ResponseFormatter {

  static success(res, data, statusCode = 200, metadata = {}) {

    const response = {

      success: true,

      data: data,

      ...metadata,

      timestamp: new Date().toISOString(),

      requestId: res.locals.requestId

    };



    return res.status(statusCode).json(response);

  }


  static error(res, error, statusCode = 500) {

    const response = {

      success: false,

      error: error.code || 'INTERNAL_ERROR',

      message: error.message,

      details: error.details || [],

      timestamp: new Date().toISOString(),

      requestId: res.locals.requestId

    };



    return res.status(statusCode).json(response);

  }


  static paginated(res, data, pagination, metadata = {}) {

    return ResponseFormatter.success(res, data, 200, {

      pagination: pagination,

      ...metadata

    });

  }
```

```
}


module.exports = ResponseFormatter;
```

Core API Route Skeleton

Product catalog routes demonstrate the coordination pattern:

JAVASCRIPT

```javascript
// src/api/routes/products.js

const express = require('express');

const { body, query, param } = require('express-validator');

const ResponseFormatter = require('../middleware/response-formatter');

const ValidationMiddleware = require('../middleware/validation');


class ProductRoutes {

  constructor(productCatalog) {

    this.router = express.Router();

    this.productCatalog = productCatalog;

    this.setupRoutes();

  }


  setupRoutes() {

    // GET /api/products - Product listing with pagination and filtering

    this.router.get('/',

      [

        query('page').optional().isInt({ min: 1 }),

        query('limit').optional().isInt({ min: 1, max: 100 }),

        query('sort').optional().isIn(['name_asc', 'name_desc', 'price_asc', 'price_desc']),

        query('category').optional().isInt({ min: 1 }),

        query('minPrice').optional().isInt({ min: 0 }),

        query('maxPrice').optional().isInt({ min: 0 })

      ],

      ValidationMiddleware.handleValidationErrors,

      this.listProducts.bind(this)

    );


    // GET /api/products/search - Product search

    this.router.get('/search',

      [

        query('q').notEmpty().withMessage('Search query is required'),

        query('page').optional().isInt({ min: 1 }),
```

```
      query('limit').optional().isInt({ min: 1, max: 100 })
    ],
    ValidationMiddleware.handleValidationErrors,
    this.searchProducts.bind(this)
  );

  // GET /api/products/:id - Individual product details
  this.router.get('/:id',
    [
      param('id').isInt({ min: 1 }).withMessage('Product ID must be positive integer')
    ],
    ValidationMiddleware.handleValidationErrors,
    this.getProduct.bind(this)
  );
}

async listProducts(req, res, next) {
  try {
    // TODO 1: Extract query parameters with defaults
    // TODO 2: Build filter criteria object
    // TODO 3: Call productCatalog.findWithFilters()
    // TODO 4: Format response with pagination metadata
    // TODO 5: Return formatted ProductListResponse
  } catch (error) {
    next(error);
  }
}

async searchProducts(req, res, next) {
  try {
    // TODO 1: Extract and validate search query
    // TODO 2: Parse additional filters from query params
    // TODO 3: Call productCatalog.searchProducts()
```

```
      // TODO 4: Calculate search result metadata (result count, suggestions)
      // TODO 5: Return formatted search results with metadata
    } catch (error) {
      next(error);
    }
  }

  async getProduct(req, res, next) {
    try {
      // TODO 1: Extract product ID from params
      // TODO 2: Call productCatalog.findById()
      // TODO 3: Handle not found case (404 response)
      // TODO 4: Format product detail response
      // TODO 5: Include related products or recommendations
    } catch (error) {
      next(error);
    }
  }
}

module.exports = ProductRoutes;
```

Cart Coordination Route Skeleton

Shopping cart routes demonstrate session-based state coordination:

JAVASCRIPT

```javascript
// src/api/routes/cart.js

const express = require('express');

const { body, param } = require('express-validator');

const AuthenticationMiddleware = require('../middleware/auth');

const ResponseFormatter = require('../middleware/response-formatter');


class CartRoutes {

  constructor(cartManager, productCatalog) {

    this.router = express.Router();

    this.cartManager = cartManager;

    this.productCatalog = productCatalog;

    this.setupRoutes();

  }


  setupRoutes() {

    // All cart routes require session (authenticated or anonymous)

    this.router.use(AuthenticationMiddleware.validateSession);


    // GET /api/cart - Get current cart contents

    this.router.get('/', this.getCart.bind(this));


    // POST /api/cart/items - Add item to cart

    this.router.post('/items',

      [

        body('productId').isInt({ min: 1 }).withMessage('Valid product ID required'),

        body('quantity').isInt({ min: 1 }).withMessage('Quantity must be positive')

      ],

      this.addToCart.bind(this)

    );


    // PUT /api/cart/items/:id - Update cart item quantity

    this.router.put('/items/:id',

      [

        param('id').isInt({ min: 1 }),
```

```
      body('quantity').isInt({ min: 1 })
    ],
    this.updateCartItem.bind(this)
  );

  // DELETE /api/cart/items/:id - Remove item from cart
  this.router.delete('/items/:id',
    [param('id').isInt({ min: 1 })],
    this.removeFromCart.bind(this)
  );
}

async getCart(req, res, next) {
  try {
    // TODO 1: Get session ID from req.session
    // TODO 2: Call cartManager.getCartContents()
    // TODO 3: Calculate cart totals and metadata
    // TODO 4: Check for price changes since items added
    // TODO 5: Return CartSummary response
  } catch (error) {
    next(error);
  }
}

async addToCart(req, res, next) {
  try {
    // TODO 1: Extract productId and quantity from request body
    // TODO 2: Validate product exists using productCatalog.findById()
    // TODO 3: Check inventory availability
    // TODO 4: Call cartManager.addItem() with session context
    // TODO 5: Return updated CartSummary
    // Hint: Handle inventory conflicts with 409 status code
  } catch (error) {
```

```
      next(error);
    }
  }

  async updateCartItem(req, res, next) {
    try {
      // TODO 1: Extract cart item ID and new quantity
      // TODO 2: Validate cart item belongs to current session
      // TODO 3: Check inventory for new quantity
      // TODO 4: Call cartManager.updateQuantity()
      // TODO 5: Return updated cart summary
    } catch (error) {
      next(error);
    }
  }

  async removeFromCart(req, res, next) {
    try {
      // TODO 1: Extract cart item ID from params
      // TODO 2: Verify item belongs to current session
      // TODO 3: Call cartManager.removeItem()
      // TODO 4: Return updated cart summary
      // TODO 5: Handle item not found with 404
    } catch (error) {
      next(error);
    }
  }
}

module.exports = CartRoutes;
```

Main Application Assembly

The main application file coordinates all components and establishes the request processing pipeline:

JAVASCRIPT

```javascript
// src/app.js

const express = require('express');

const cookieParser = require('cookie-parser');

const cors = require('cors');


// Component imports

const Database = require('./database/Database');

const ProductCatalog = require('./components/ProductCatalog');

const CartManager = require('./components/CartManager');

const Authentication = require('./components/Authentication');

const CheckoutProcessor = require('./components/CheckoutProcessor');


// Route imports

const ProductRoutes = require('./api/routes/products');

const CartRoutes = require('./api/routes/cart');

const AuthRoutes = require('./api/routes/auth');

const CheckoutRoutes = require('./api/routes/checkout');


class EcommerceApp {

  constructor() {

    this.app = express();

    this.database = null;

    this.components = {};

  }


  async initialize() {

    // TODO 1: Initialize database connection

    // TODO 2: Create and initialize all business logic components

    // TODO 3: Setup Express middleware stack

    // TODO 4: Mount API routes with component dependencies

    // TODO 5: Setup error handling middleware

  }


  setupMiddleware() {
```

```
    // TODO 1: Configure CORS for frontend domain
    // TODO 2: Setup body parsing for JSON payloads
    // TODO 3: Configure cookie parsing for sessions
    // TODO 4: Add request ID generation for debugging
    // TODO 5: Setup request logging middleware
  }

  setupRoutes() {
    // TODO 1: Create route instances with component dependencies
    // TODO 2: Mount routes under /api prefix
    // TODO 3: Setup 404 handler for unknown routes
    // TODO 4: Add health check endpoint
  }

  setupErrorHandling() {
    // TODO 1: Create centralized error handler
    // TODO 2: Handle validation errors specifically
    // TODO 3: Handle database connection errors
    // TODO 4: Setup error logging
    // TODO 5: Return consistent error response format
  }

  async start(port = 3000) {
    await this.initialize();

    this.app.listen(port, () => {
      console.log(`E-commerce API server running on port ${port}`);
      console.log(`Health check: http://localhost:${port}/health`);
    });
  }

  async shutdown() {
    // TODO 1: Close database connections gracefully
    // TODO 2: Finish processing in-flight requests
```

```
    // TODO 3: Clear any scheduled cleanup tasks
  }
}

module.exports = EcommerceApp;
```

Milestone Checkpoints

After implementing the interaction layer for each milestone:

Milestone 1 Checkpoint (Product Catalog Interactions):

- Start server with `node src/app.js`
- Test product listing: `curl http://localhost:3000/api/products?page=1&limit=5`
- Expected: JSON response with products array and pagination metadata
- Test product search: `curl http://localhost:3000/api/products/search?q=laptop`
- Expected: Filtered products matching search term
- Test individual product: `curl http://localhost:3000/api/products/1`
- Expected: Single product with full details

Milestone 2 Checkpoint (Cart Interactions):

- Test anonymous cart creation: `curl -c cookies.txt -X POST http://localhost:3000/api/cart/items -H "Content-Type: application/json" -d '{"productId": 1, "quantity": 2}'`
- Expected: 201 status with CartSummary response, session cookie set
- Test cart retrieval: `curl -b cookies.txt http://localhost:3000/api/cart`
- Expected: Cart contents with items, totals, and metadata

Milestone 3 Checkpoint (Authentication Interactions):

- Test user registration: `curl -X POST http://localhost:3000/api/auth/register -H "Content-Type: application/json" -d '{"email": "test@example.com", "password": "SecurePass123!"}'`
- Expected: 201 status with user profile, session cookie set
- Test login: `curl -c auth-cookies.txt -X POST http://localhost:3000/api/auth/login -H "Content-Type: application/json" -d '{"email": "test@example.com", "password": "SecurePass123!"}'`
- Expected: 200 status with user profile and valid session

Milestone 4 Checkpoint (Checkout Interactions):

- Test checkout start: `curl -b auth-cookies.txt -X POST http://localhost:3000/api/checkout/start`
- Expected: 201 status with checkout session ID
- Test address collection: `curl -b auth-cookies.txt -X PUT http://localhost:3000/api/checkout/{id}/address -H "Content-Type: application/json" -d '{"shippingAddress": {...}}'`
- Expected: 200 status with updated checkout session

Debugging Common Integration Issues

Symptom	Likely Cause	Diagnosis	Fix
"Cannot set headers after they are sent"	Multiple response calls in route handler	Check for missing return statements before res.json()	Add return before response calls
Session not persisting across requests	Cookie configuration or parsing issues	Check browser dev tools for cookie presence	Verify cookie-parser middleware and domain settings
500 errors on valid requests	Unhandled promise rejections	Check server logs for stack traces	Add try-catch blocks and error middleware
Validation errors not showing details	Error handling middleware not formatting properly	Test error responses with invalid data	Implement detailed error response structure
Cart operations failing silently	Session/user context not available in routes	Add logging to verify session data in req.session	Ensure auth middleware runs before cart routes

Error Handling and Edge Cases

> **Milestone(s):** All milestones - defines failure modes, error detection strategies, and recovery mechanisms that span product catalog (Milestone 1), shopping cart (Milestone 2), user authentication (Milestone 3), and checkout process (Milestone 4)

Building reliable e-commerce systems requires anticipating and gracefully handling countless failure scenarios. Think of error handling like designing a safety net for a trapeze act - you hope the performers never fall, but when they do, the net must catch them safely and allow the show to continue. In e-commerce, "falling" means inventory conflicts, payment failures, network timeouts, and concurrent user actions that could leave the system in an inconsistent state. The safety net is a comprehensive error handling strategy that detects problems early, recovers gracefully, and preserves user experience even when things go wrong.

E-commerce systems are particularly challenging because they involve real money, finite inventory, and user expectations of immediate responsiveness. Unlike a blog or content site where a temporary error might be merely annoying, e-commerce errors can mean lost sales, oversold inventory, or frustrated customers abandoning their carts. The key insight is that **error handling is not just about preventing crashes - it's about maintaining business continuity and user trust**.

The complexity arises from the intersection of multiple failure domains: database constraints that prevent overselling, network timeouts during payment processing, race conditions when multiple users purchase the last item, session expiration during checkout, and external service failures like payment gateways or shipping calculators. Each component must handle its own failures while coordinating with other components to maintain overall system consistency.

Failure Modes

Inventory Consistency Failures represent one of the most critical failure modes in e-commerce systems. Think of inventory like concert tickets - when the venue sells out, you cannot create more seats just because customers keep clicking "buy now". The fundamental challenge is that **inventory is a finite, shared resource** that must be managed consistently across concurrent operations.

The most common inventory failure mode occurs when multiple users attempt to purchase the same product simultaneously, particularly when inventory levels are low. Consider this scenario: Product A has 1 unit remaining in inventory. User X and User Y both view the product page, see "1 available", and click "Add to Cart" within milliseconds of each other. Without proper coordination, both carts could show the item as available, leading to an oversell situation during checkout.

Failure Mode	Trigger Condition	Data Impact	User Experience Impact
Race Condition Oversell	Concurrent add-to-cart operations on low inventory	Negative inventory quantities in database	Multiple users believe they can purchase unavailable items
Cart-Checkout Inventory Drift	Inventory changes between cart addition and checkout	Cart contains unavailable items	Checkout fails unexpectedly with inventory errors
Abandoned Cart Phantom Reservations	Items added to cart but never purchased	Inventory appears lower than actual availability	False scarcity prevents valid purchases
Price Change Conflicts	Product prices update while items remain in user carts	Carts show outdated pricing information	Price discrepancies at checkout confuse users
Session-Based Inventory Leaks	User sessions expire with items still in cart	Inventory remains falsely reserved	Available inventory appears lower than reality

Payment Processing Failures introduce another layer of complexity because they involve external systems beyond our direct control. Payment gateways can experience network issues, decline transactions for fraud prevention, or suffer temporary outages. The critical insight is that **payment failures can occur at multiple stages**: authorization, capture, or settlement.

During the authorization phase, the payment processor checks if funds are available and temporarily reserves them. This can fail due to insufficient funds, expired cards, or fraud detection triggers. During capture, the previously authorized amount is actually charged to the customer's account. This can fail if the authorization has expired or if the card has been canceled. Settlement occurs when funds actually transfer from the customer's bank to the merchant account, which can fail due to banking system issues or account problems.

Payment Failure Type	Detection Point	Recovery Strategy	User Communication
Authorization Declined	Payment gateway response	Prompt for alternative payment method	"Payment declined. Please try a different card."
Authorization Timeout	No response within timeout window	Retry authorization once, then fail gracefully	"Payment processing timeout. Please try again."
Capture Failed	During order finalization	Void authorization, release inventory	"Payment processing failed. Order canceled."
Partial Capture	Split shipment scenarios	Handle remaining authorization separately	"Partial payment processed for shipped items."
Gateway Unavailable	Network or service errors	Queue order for later processing	"Payment system temporarily unavailable."

System Unavailability Failures encompass scenarios where critical dependencies become unreachable or unresponsive. Modern e-commerce platforms rely on numerous external services: payment gateways, shipping calculators, tax services, fraud detection APIs, and content delivery networks for product images. When these services fail, the application must degrade gracefully rather than completely breaking.

Database failures represent the most severe form of system unavailability because they threaten data persistence and consistency. A database connection timeout during order creation could leave the system in an inconsistent state where payment was captured but no order record exists. Network partitions can isolate the application server from the database, making it impossible to read inventory levels or update order status.

System Component	Failure Symptoms	Immediate Impact	Cascading Effects
Primary Database	Connection timeouts, query failures	Cannot read/write core data	All operations requiring persistence fail
Payment Gateway	API timeouts, HTTP 5xx errors	Cannot process payments	Checkout flow completely blocked
Image CDN	404 errors, slow responses	Product images fail to load	Poor user experience, potential lost sales
Search Service	Index unavailable, query errors	Search functionality broken	Users cannot find products effectively
Session Store	Redis/cache unavailable	User sessions lost	Users logged out, carts disappear
External APIs	Rate limiting, service outages	Tax calculation, shipping estimates fail	Checkout cannot complete order totals

Error Detection

Validation Strategies form the foundation of error detection by catching problems before they propagate through the system. Think of validation like quality control checkpoints in a manufacturing assembly line - the earlier you catch defects, the less expensive they are to fix and the less disruption they cause downstream. In e-commerce, validation occurs at multiple layers: client-side for immediate feedback, server-side for security and data integrity, and database-level for ultimate consistency enforcement.

Client-side validation provides immediate user feedback but cannot be trusted for security or data integrity because users can manipulate or bypass it. Its primary purpose is **user experience optimization** - helping users correct obvious mistakes like invalid email formats or missing required fields before they submit forms. Server-side validation is authoritative and must re-validate all client input, even if client-side validation passed.

The validation strategy employs a **defense in depth** approach where each layer catches different types of errors. Form validation catches formatting errors, business logic validation catches constraint violations, and database validation catches concurrency conflicts that could only be detected during the actual transaction.

Validation Layer	Validation Types	Performance Impact	Security Role
Client-Side	Format, required fields, basic ranges	Minimal - runs in browser	User experience only, not trusted
API Gateway	Authentication, rate limiting, request size	Low - simple checks	First line of security defense
Business Logic	Inventory constraints, pricing rules, user permissions	Medium - requires database queries	Core business rule enforcement
Database Constraints	Foreign keys, unique indexes, check constraints	High - impacts transaction performance	Final consistency guarantee

Input validation requires comprehensive checks across all data entry points. For product data, this means validating that prices are positive numbers with appropriate precision, inventory quantities are non-negative integers, and category assignments reference valid categories. For user data, validation includes email format checking, password strength requirements, and address field completeness for shipping calculations.

The challenge with input validation is balancing thoroughness with performance. Comprehensive validation requires database queries to check foreign key relationships and business constraints, but these queries add latency to user operations. The solution is **layered validation** that performs expensive checks only when necessary and caches validation results when possible.

Input Type	Validation Rules	Error Messages	Recovery Actions
Product Price	Positive decimal, max 2 decimal places	"Price must be a positive amount with cents precision"	Prompt for corrected price entry
Email Address	RFC 5322 format, domain validation	"Please enter a valid email address"	Highlight invalid field, suggest corrections
Inventory Quantity	Non-negative integer, reasonable maximum	"Inventory must be a whole number"	Reset to previous valid value
Credit Card	Luhn algorithm, expiration date future	"Invalid card number or expired card"	Prompt for card re-entry
Shipping Address	Required fields, postal code format	"Complete address required for shipping"	Highlight missing fields

Monitoring Approaches provide early warning systems that detect problems before they impact users or escalate into major failures. Effective monitoring combines **proactive health checks** with **reactive error tracking** to create comprehensive visibility into system behavior. Think of monitoring like a medical examination that combines routine vital signs (proactive) with symptom investigation when problems arise (reactive).

Health checks continuously verify that critical system components are functioning correctly. These automated checks simulate user operations like product search, cart operations, and authentication flows to detect problems immediately when they occur. The monitoring system must distinguish between transient issues that resolve automatically and persistent problems requiring intervention.

Error tracking captures and analyzes actual failures as they happen in production. This includes application exceptions, database constraint violations, payment failures, and external service timeouts. The key insight is that **error patterns often reveal system design problems** that health checks cannot detect because they only become apparent under specific load conditions or data combinations.

Monitoring Type	Check Frequency	Alert Thresholds	Response Actions
Database Health	Every 30 seconds	Query time > 5s, connection failures	Auto-restart connection pool, notify ops team
Payment Gateway	Every 2 minutes	Error rate > 5%, response time > 10s	Switch to backup gateway, escalate to vendor
Inventory Consistency	Every 5 minutes	Negative inventory detected	Lock affected products, trigger reconciliation
Session Store	Every 1 minute	Redis unavailable, high memory usage	Failover to backup instance, clear old sessions
API Response Times	Real-time	95th percentile > 2s	Enable performance logging, check database
Error Rates	Real-time	Error rate > 2%	Alert development team, check recent deployments

Business Logic Validation ensures that operations comply with domain-specific rules that go beyond simple data format checking. In e-commerce, business logic validation includes rules like "users cannot purchase more items than available inventory", "discount codes must be valid and not expired", and "shipping addresses must be deliverable locations". These validations often require complex queries across multiple database tables and may involve external service calls.

The challenge with business logic validation is that it must be **consistent across all entry points** while remaining **performant under load**. The same inventory validation logic must work identically whether a user adds items through the web interface, mobile

app, or administrative system. This requires centralizing validation logic in shared components that all interfaces can use.

Business Rule	Validation Query	Performance Optimization	Error Handling
Inventory Availability	`SELECT inventory_quantity FROM products WHERE id = ?`	Cache frequently checked products	Reserve inventory during validation
User Purchase Limits	`SELECT SUM(quantity) FROM orders WHERE user_id = ? AND created_at > ?`	Index on user_id and created_at	Explain limits clearly to user
Discount Code Validity	`SELECT * FROM discount_codes WHERE code = ? AND expires_at > NOW()`	Index on code and expiration	Suggest similar valid codes
Address Deliverability	External geocoding API call	Cache validation results by address	Offer address correction suggestions
Payment Method Active	Query payment processor API	Cache active status briefly	Prompt for alternative payment

Recovery Strategies

Graceful Degradation represents the art of **failing gracefully** when systems cannot operate at full capacity. Think of graceful degradation like a restaurant during a power outage - they cannot use electric appliances, but they can still serve cold dishes, use manual cash registers, and provide candlelit dining. The restaurant continues operating with reduced functionality rather than closing completely. In e-commerce systems, graceful degradation means maintaining core shopping functionality even when auxiliary services fail.

The key principle is **progressive functionality reduction** that prioritizes essential operations over convenience features. When the search service fails, users can still browse products by category. When the recommendation engine fails, users can still find products manually. When external shipping calculators fail, the system can use default shipping rates. The goal is to **preserve revenue-generating functionality** even when the user experience is suboptimal.

Graceful degradation requires **design-time decisions** about which features are essential versus optional. These decisions must be embedded in the system architecture so that failures automatically trigger appropriate fallback behavior without manual intervention. The system must also communicate clearly to users when functionality is limited and suggest alternative paths to accomplish their goals.

Failed Component	Essential Functionality Preserved	Graceful Degradation Strategy	User Communication
Product Search Service	Category browsing, direct product URLs	Show category navigation, disable search box	"Search temporarily unavailable. Browse by category."
Payment Gateway	Alternative payment methods	Redirect to backup processor	"Primary payment method unavailable. Try PayPal."
Shipping Calculator	Flat rate shipping estimates	Use default shipping rates	"Exact shipping calculated at delivery."
Recommendation Engine	Manual product discovery	Hide recommendation sections	Remove "You may also like" widgets silently
Image CDN	Text-based product information	Show placeholder images with descriptions	Display "Image loading..." with product details
External Tax API	Estimated tax calculations	Use default tax rates by region	"Tax estimate - final amount may vary slightly."

Compensating Transactions provide a mechanism for **undoing partially completed operations** when later steps in a multi-step process fail. Unlike database transactions that can be simply rolled back, business operations often involve external systems that cannot be easily reversed. Think of compensating transactions like a bank's process for reversing a mistaken transfer - they cannot simply "undo" the original transaction, but they can execute a counteracting transaction that restores the correct state.

In e-commerce, compensating transactions become crucial during checkout operations that involve multiple steps: inventory reservation, payment authorization, order creation, and notification sending. If payment authorization succeeds but order creation fails, the system must execute a compensating transaction to void the payment authorization and release the reserved inventory.

The challenge with compensating transactions is **ensuring they actually restore the correct state** even when executed out of order or partially. Compensating transactions must be **idempotent** (safe to execute multiple times) and **commutative** (safe to execute in any order) to handle retry scenarios and recovery edge cases.

Original Operation	Failure Point	Compensating Transaction	State Restoration
Reserve Inventory	Payment authorization fails	`UPDATE products SET reserved_qty = reserved_qty - ? WHERE id = ?`	Returns inventory to available pool
Authorize Payment	Order creation fails	`POST /payments/{auth_id}/void` to payment gateway	Releases customer's funds from hold
Create Order Record	Inventory update fails	`DELETE FROM orders WHERE id = ?` and restore cart	Removes order, restores shopping cart
Send Confirmation Email	Email service unavailable	Queue email for retry, mark order as "pending notification"	Customer receives confirmation when service recovers
Update User Points	Points calculation error	`UPDATE users SET loyalty_points = loyalty_points - ? WHERE id = ?`	Reverses incorrect points addition
Create Shipping Label	Shipping service fails	Cancel order, process refund	Returns to pre-order state with payment reversal

Data Consistency Preservation ensures that the system maintains accurate and coherent information even when individual operations fail. Think of data consistency like maintaining accurate financial books during a busy day at a retail store - every sale

must be recorded correctly, inventory must be decremented, and cash received must match receipts, even when the cash register occasionally jams or the credit card reader fails.

The fundamental challenge is that **e-commerce operations often span multiple data entities** that must remain synchronized. When a customer completes a purchase, the system must update inventory quantities, create order records, modify user account information, and potentially adjust promotional usage counters. If any of these updates fail, the system must either complete all updates or revert all changes to maintain consistency.

ACID transaction principles provide the foundation for data consistency preservation: **Atomicity** ensures all changes succeed or fail together, **Consistency** ensures all database constraints remain valid, **Isolation** prevents concurrent operations from interfering with each other, and **Durability** ensures committed changes survive system failures.

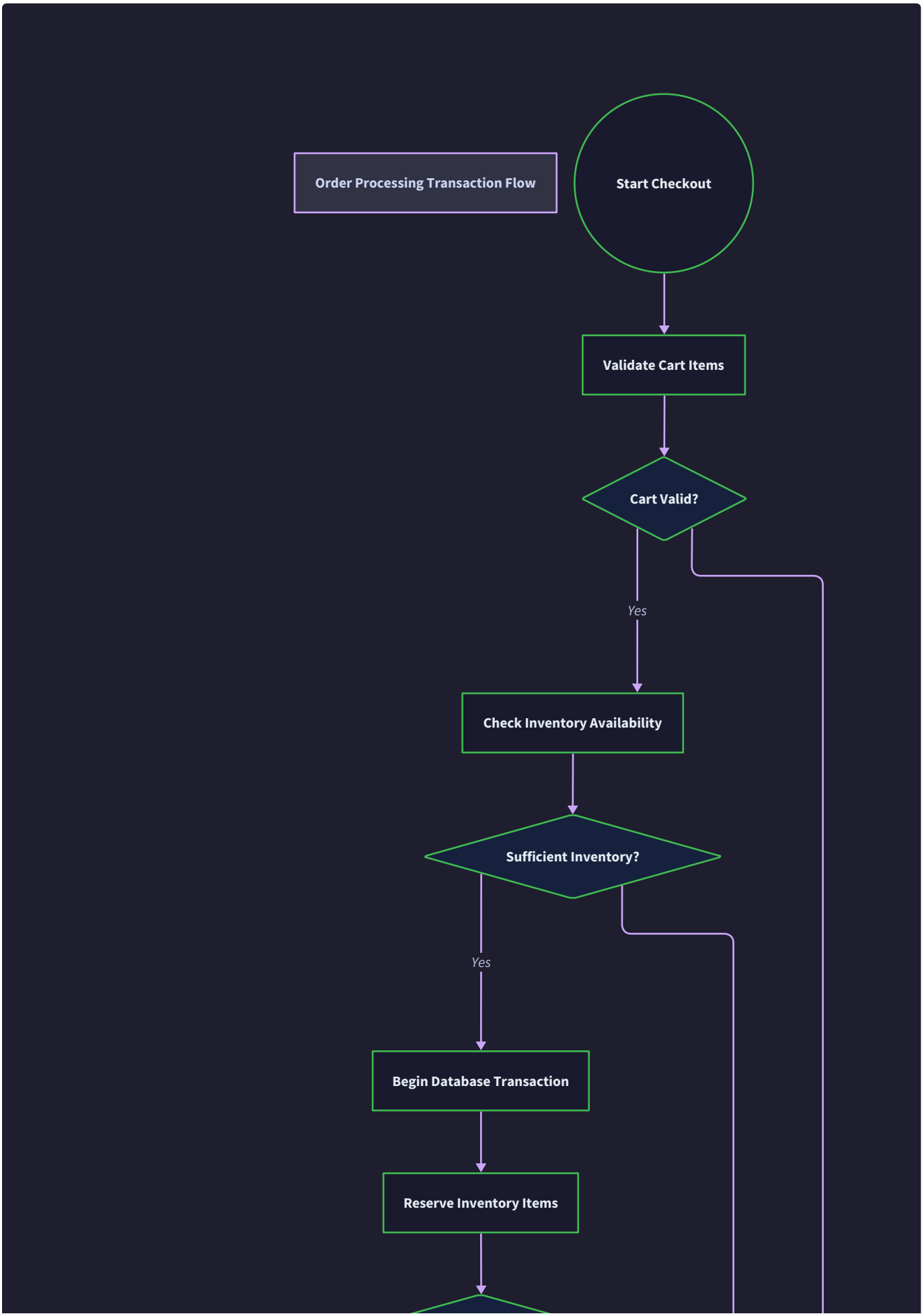
Consistency Requirement	Implementation Strategy	Conflict Detection	Resolution Method
Inventory Accuracy	Pessimistic locking during checkout using `SELECT FOR UPDATE`	Compare expected vs actual inventory before commit	Abort transaction, notify user of unavailability
Order Completeness	Single database transaction for all order-related updates	Foreign key constraints prevent orphaned records	Rollback entire transaction on any failure
Price Integrity	Store price snapshot in cart items and order items	Compare cart price vs current product price	Allow checkout with notification of price change
User Account Balance	Atomic increment/decrement operations	Check account balance before deducting points/credits	Reject operation if insufficient balance
Session Consistency	Store cart contents in database linked to session	Validate session exists before cart operations	Redirect to login if session invalid
Payment Status Accuracy	Store payment gateway transaction ID with order	Query payment gateway for actual transaction status	Update order status to match gateway records

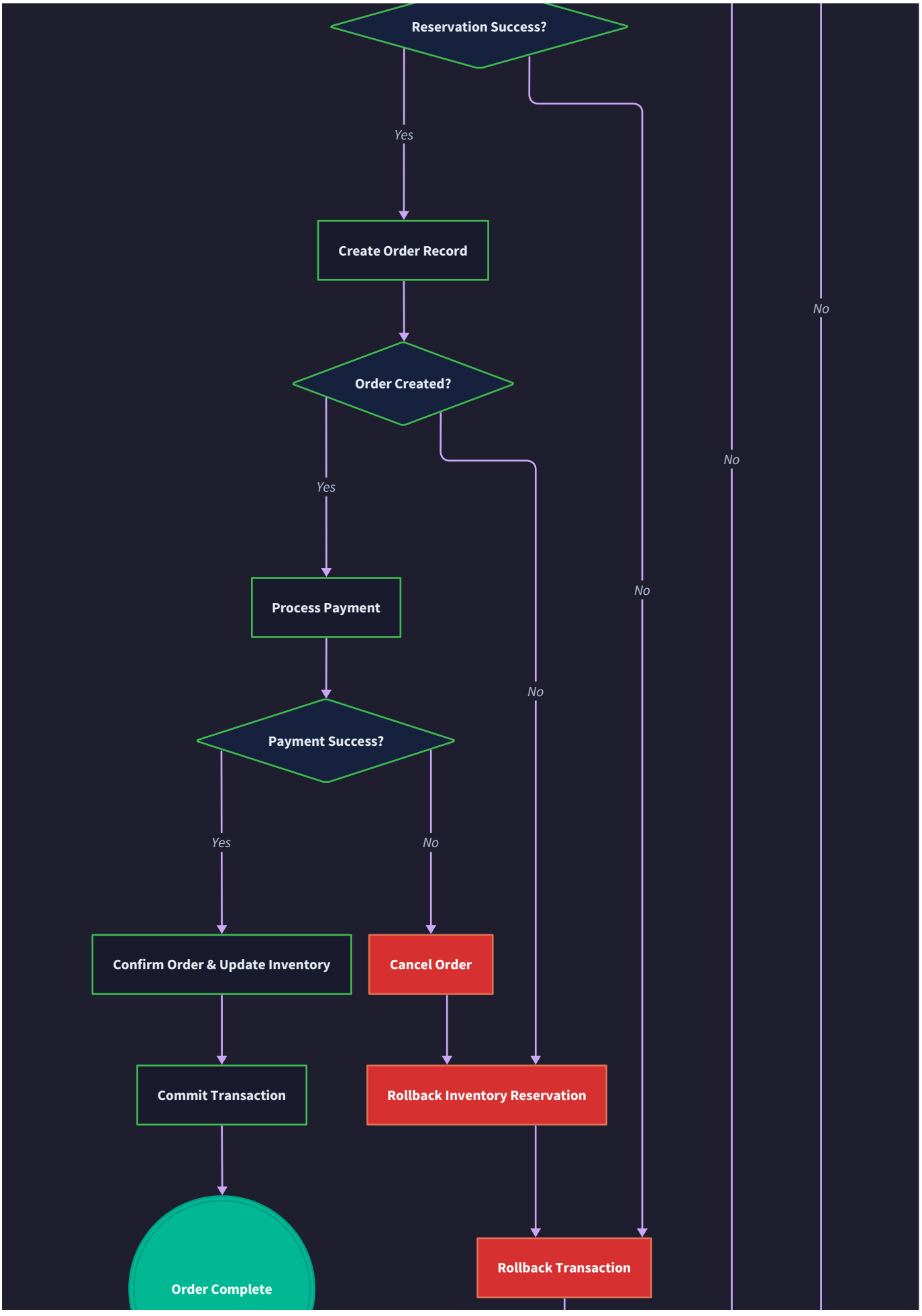
User Experience Preservation focuses on **maintaining customer satisfaction** even when technical problems occur behind the scenes. The goal is to shield users from system complexity while providing clear guidance about what happened and what they should do next. Think of this like a skilled hotel concierge handling a problem - they apologize for the inconvenience, explain the situation clearly, offer concrete alternatives, and follow up to ensure satisfaction.

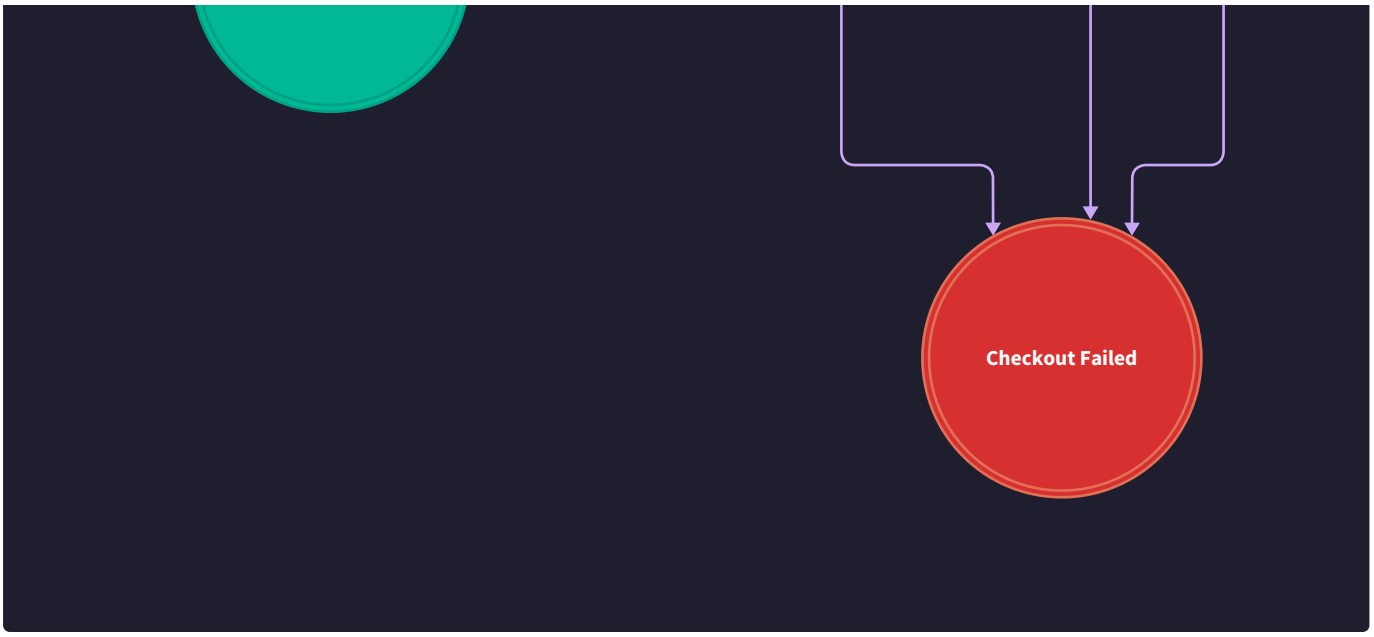
The key insight is that **users care about accomplishing their goals, not about technical system details**. When inventory runs out during checkout, users need to know immediately and be offered similar products. When payment processing fails, users need to know their card was not charged and be guided toward alternative payment methods. When the system experiences temporary issues, users need realistic estimates of when functionality will be restored.

Error messages must be **actionable and specific** rather than generic or technical. Instead of "An error occurred while processing your request", effective messages explain what went wrong and what the user should do: "This item is no longer available. We've saved your cart and found similar products you might like." The message provides context, reassurance, and a clear path forward.

Error Scenario	Technical Root Cause	User-Facing Message	Recovery Actions Offered
Inventory Conflict	Race condition during checkout	"Sorry, this item just sold out. We've saved your cart."	Show similar products, offer back-in-stock notifications
Payment Decline	Card declined by processor	"Payment could not be processed. Your card was not charged."	Suggest trying a different card, offer PayPal option
Session Expired	User idle during checkout	"Your session expired for security. Please log in to continue."	Preserve cart contents, streamline re-authentication
Price Change	Product price updated while in cart	"The price of [item] has changed from $X to $Y."	Allow checkout at new price, option to remove item
Shipping Error	Address validation failed	"We need to verify your shipping address."	Highlight problematic fields, suggest corrections
System Maintenance	Planned downtime	"Site maintenance in progress. Back online in 30 minutes."	Allow browsing, disable checkout, show countdown timer


Order Processing Transaction Flow
Start Checkout
Validate Cart Items
Cart Valid?
Yes
Check Inventory Availability
Sufficient Inventory?
Yes
Begin Database Transaction
Reserve Inventory Items


Reservation Success?
Yes
No
Create Order Record
Order Created?
Yes
No
Process Payment
Payment Success?
Yes
No
Confirm Order & Update Inventory
Cancel Order
Commit Transaction
Rollback Inventory Reservation
Order Complete
Rollback Transaction
No
No
No



Transaction Rollback Mechanisms provide the ability to **atomically undo complex operations** that involve multiple database changes. In e-commerce systems, rollback becomes critical when order processing encounters errors after some changes have already been committed. Think of transaction rollback like a video game's save/restore feature - when something goes wrong, you can return to a known good state and try a different approach.

The challenge in e-commerce is that operations often **cross transaction boundaries** because they involve external systems like payment gateways that cannot participate in database transactions. This requires implementing **application-level coordination** that combines database transactions with compensating actions for external systems.

Effective rollback mechanisms require **careful transaction design** that minimizes the scope of changes while maintaining data integrity. Long-running transactions increase the likelihood of conflicts and failures, so the strategy is to break complex operations into smaller, independent transactions with explicit coordination points.

Transaction Scope	Rollback Trigger	Rollback Actions	State Verification
Single Product Purchase	Payment authorization fails	Rollback inventory reservation, release cart lock	Verify inventory matches pre-transaction state
Multi-Item Order	Insufficient inventory for any item	Rollback all inventory changes, void payment authorization	Confirm no partial inventory updates occurred
User Account Update	Email sending fails after account creation	Rollback user creation, return registration form	Verify user account does not exist in system
Bulk Inventory Update	Constraint violation on any product	Rollback all product updates in batch	Confirm inventory matches pre-update snapshot
Price Change Batch	Any price validation failure	Rollback all price changes in batch	Verify prices match previous values
Order Modification	Shipping calculation fails	Restore original order state	Confirm order matches pre-modification version

> **Architecture Decision: Error Handling Strategy**
>
> **Context**: E-commerce systems must handle failures gracefully while maintaining data consistency and user trust. Errors can occur at multiple layers (client, server, database, external services) and may be transient or persistent. The system must distinguish between recoverable and non-recoverable failures while providing appropriate user feedback.
>
> **Options Considered**:
>
> 1. **Fail Fast Strategy**: Immediately abort operations on any error and return error messages to users
> 2. **Retry-Heavy Strategy**: Automatically retry all failed operations multiple times before giving up
> 3. **Graceful Degradation Strategy**: Continue operating with reduced functionality when non-critical components fail
>
> **Decision**: Implement a **Graceful Degradation Strategy** with intelligent retry logic and comprehensive compensating transactions.
>
> **Rationale**: E-commerce systems must prioritize business continuity and revenue preservation over perfect functionality. Users will tolerate temporarily degraded features (like missing product recommendations) but will abandon the platform if core shopping functionality fails completely. Automatic retries help with transient failures, but graceful degradation ensures the system remains usable even during extended outages of auxiliary services.
>
> **Consequences**: This approach requires more complex error handling logic and careful feature prioritization, but it provides better user experience and business continuity. It also requires comprehensive monitoring to detect when the system is operating in degraded mode.

Common Pitfalls

⚠️ Pitfall: Ignoring Race Conditions in Inventory Management

Many developers implement inventory checking as a simple query followed by an update, not realizing that concurrent operations can create race conditions that lead to overselling. The typical mistake is checking inventory availability in one database query, then updating it in a separate query without proper locking. Between these two queries, another user might purchase the same item, leading to negative inventory.

```
// WRONG: Race condition possible
SELECT inventory_quantity FROM products WHERE id = 123;  -- Returns 1
-- Another user's transaction decrements inventory to 0 here
UPDATE products SET inventory_quantity = 0 WHERE id = 123;  -- Creates oversell
```

The solution is to use pessimistic locking with `SELECT FOR UPDATE` to ensure atomic inventory checks and updates within a single transaction. This prevents concurrent modifications during the critical inventory validation period.

⚠️ Pitfall: Not Implementing Compensating Transactions for External Services

When checkout involves external services like payment gateways, developers often forget that these operations cannot be rolled back with simple database transactions. If payment authorization succeeds but order creation fails, the customer's funds remain on hold indefinitely without any order record to track the authorization.

The fix requires implementing explicit compensating transactions that void payment authorizations when subsequent operations fail. This must be handled at the application level since payment gateways cannot participate in database transactions.

⚠️ Pitfall: Exposing Technical Error Details to Users

Developers frequently display raw database errors or exception messages directly to users, which provides no actionable information and may expose sensitive system details. Error messages like "Foreign key constraint violation in table 'orders'" confuse users and reveal internal system structure.

Effective error handling translates technical errors into user-friendly messages with specific recovery actions. Every error scenario should have a predefined user message that explains what happened and what the user should do next.

⚠️ Pitfall: Insufficient Session Cleanup Leading to Memory Leaks

E-commerce systems often create session data for cart contents and user preferences but fail to implement proper cleanup when sessions expire or users abandon their carts. Over time, this creates memory leaks in session stores and database bloat from orphaned cart records.

The solution requires implementing automatic session cleanup with configurable timeouts and periodic cleanup jobs that remove expired sessions and abandoned carts.

⚠️ Pitfall: Not Testing Error Scenarios Under Load

Many error handling mechanisms work correctly under normal conditions but fail catastrophically under high load when multiple users trigger the same error conditions simultaneously. Error handling code paths are often untested because they are difficult to reproduce consistently in development environments.

Comprehensive testing requires deliberately triggering error conditions during load testing to verify that error handling mechanisms scale appropriately and don't create additional bottlenecks when the system is already stressed.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Error Logging	Winston with file transport	ELK Stack (Elasticsearch, Logstash, Kibana)
Monitoring	Basic health check endpoints	Prometheus + Grafana with custom metrics
Session Management	Express-session with memory store	Redis-based session store with clustering
Database Transactions	SQLite with manual transaction handling	PostgreSQL with automatic retry and deadlock detection
Payment Processing	Mock payment gateway for development	Stripe with webhook handling
Circuit Breaker	Simple retry logic with exponential backoff	Netflix Hystrix pattern implementation

Recommended File Structure:

```
src/
├── middleware/
│   ├── errorHandler.js          ← centralized error handling
│   ├── validation.js            ← input validation middleware
│   └── circuitBreaker.js        ← external service protection
├── services/
│   ├── inventoryService.js      ← inventory consistency logic
│   ├── paymentService.js        ← payment processing with retries
│   └── orderService.js          ← order creation with rollback
├── utils/
│   ├── logger.js                ← structured logging configuration
│   ├── monitoring.js            ← health checks and metrics
│   └── gracefulShutdown.js      ← cleanup on application termination
├── errors/
│   ├── AppError.js              ← base error class hierarchy
│   ├── ValidationError.js       ← input validation errors
│   ├── InventoryError.js        ← inventory-specific errors
│   └── PaymentError.js          ← payment processing errors
└── config/
    └── errorConfig.js           ← error handling configuration
```

Error Handler Infrastructure (Complete):

JAVASCRIPT

```javascript
// errors/AppError.js - Base error class for application errors
class AppError extends Error {
    constructor(message, statusCode = 500, code = 'INTERNAL_ERROR', details = {}) {
        super(message);
        this.statusCode = statusCode;
        this.code = code;
        this.details = details;
        this.timestamp = new Date().toISOString();
        this.operational = true; // Distinguishes from programming errors

        Error.captureStackTrace(this, this.constructor);
    }

    toJSON() {
        return {
            message: this.message,
            code: this.code,
            statusCode: this.statusCode,
            details: this.details,
            timestamp: this.timestamp
        };
    }
}

// errors/ValidationError.js - Input validation specific errors
class ValidationError extends AppError {
    constructor(field, value, constraint) {
        const message = `Validation failed for ${field}: ${constraint}`;
        super(message, 400, 'VALIDATION_ERROR', { field, value, constraint });
    }
}

// errors/InventoryError.js - Inventory consistency errors
```

```
class InventoryError extends AppError {
    constructor(productId, requested, available) {
        const message = `Insufficient inventory for product ${productId}`;
        super(message, 409, 'INVENTORY_UNAVAILABLE', {
            productId,
            requested,
            available
        });
    }
}

// middleware/errorHandler.js - Centralized error processing
function errorHandler(err, req, res, next) {
    // Log error for monitoring and debugging
    logger.error('Request error occurred', {
        error: err.message,
        stack: err.stack,
        code: err.code,
        url: req.url,
        method: req.method,
        userId: req.session?.userId,
        timestamp: new Date().toISOString()
    });

    // Handle different error types appropriately
    if (err.operational) {
        // Known application errors with user-friendly messages
        return res.status(err.statusCode).json({
            success: false,
            error: {
                message: err.message,
                code: err.code,
```

```
                details: err.details
            }
        });
    } else {
        // Programming errors should not expose internal details
        return res.status(500).json({
            success: false,
            error: {
                message: 'An unexpected error occurred',
                code: 'INTERNAL_ERROR'
            }
        });
    }
}

module.exports = { AppError, ValidationError, InventoryError, errorHandler };
```

Circuit Breaker Implementation (Complete):

JAVASCRIPT

```javascript
// middleware/circuitBreaker.js - Protects external service calls

class CircuitBreaker {

    constructor(options = {}) {

        this.failureThreshold = options.failureThreshold || 5;

        this.resetTimeout = options.resetTimeout || 60000; // 1 minute

        this.monitoringPeriod = options.monitoringPeriod || 10000; // 10 seconds



        this.state = 'CLOSED'; // CLOSED, OPEN, HALF_OPEN

        this.failureCount = 0;

        this.lastFailureTime = null;

        this.successCount = 0;

    }



    async execute(operation, fallback) {

        if (this.state === 'OPEN') {

            if (Date.now() - this.lastFailureTime >= this.resetTimeout) {

                this.state = 'HALF_OPEN';

                this.successCount = 0;

            } else {

                return fallback ? await fallback() : null;

            }

        }



        try {

            const result = await operation();

            this.onSuccess();

            return result;

        } catch (error) {

            this.onFailure();

            if (fallback) {

                return await fallback();

            } else {
```

```
            throw error;
        }
    }
}

onSuccess() {
    if (this.state === 'HALF_OPEN') {
        this.successCount++;
        if (this.successCount >= 3) {
            this.state = 'CLOSED';
            this.failureCount = 0;
        }
    } else {
        this.failureCount = Math.max(0, this.failureCount - 1);
    }
}

onFailure() {
    this.failureCount++;
    this.lastFailureTime = Date.now();

    if (this.failureCount >= this.failureThreshold) {
        this.state = 'OPEN';
    }
}

getState() {
    return {
        state: this.state,
        failureCount: this.failureCount,
        lastFailureTime: this.lastFailureTime
    };
```

```
    }
}

module.exports = CircuitBreaker;
```

Core Error Recovery Skeleton (TODOs only):

JAVASCRIPT

```javascript
// services/orderService.js - Order creation with comprehensive error handling

class OrderService {

    constructor(db, paymentService, inventoryService, logger) {

        this.db = db;

        this.paymentService = paymentService;

        this.inventoryService = inventoryService;

        this.logger = logger;

    }



    async createOrder(cartId, shippingAddress, paymentMethod) {

        const transaction = await this.db.beginTransaction();

        let paymentAuthId = null;

        let inventoryReserved = false;



        try {

            // TODO 1: Validate cart exists and has items

            // Query cart and cart_items tables

            // Throw ValidationError if cart is empty or invalid



            // TODO 2: Reserve inventory for all cart items atomically

            // Use SELECT FOR UPDATE to lock inventory records

            // Verify sufficient inventory for each item

            // Update products.reserved_quantity for each item

            // Set inventoryReserved = true on success



            // TODO 3: Calculate order totals including tax and shipping

            // Sum cart item totals (quantity * price_snapshot)

            // Call external tax service with circuit breaker

            // Call shipping service with circuit breaker

            // Handle service failures with reasonable defaults



            // TODO 4: Authorize payment for the calculated total
```

```
            // Call paymentService.authorize() with amount and method
            // Store returned paymentAuthId for potential reversal
            // Throw PaymentError if authorization fails

            // TODO 5: Create order record with all calculated values
            // Insert into orders table with PENDING status
            // Insert order_items from cart_items with price snapshots
            // Generate unique order number for customer reference

            // TODO 6: Clear the shopping cart after successful order creation
            // Delete all cart_items for this cart
            // Update cart status to CONVERTED

            // TODO 7: Commit transaction and return order confirmation
            // Commit database transaction
            // Return OrderConfirmation with order details

        } catch (error) {
            // TODO 8: Implement comprehensive rollback on any failure
            // Rollback database transaction
            // If paymentAuthId exists, call paymentService.void(paymentAuthId)
            // If inventoryReserved, call inventoryService.releaseReservation(cartId)
            // Log error details for debugging
            // Re-throw appropriate error type for user
        }
    }

    async handleInventoryConflict(cartId, conflictedItems) {
        // TODO 1: Remove unavailable items from cart
        // Delete cart_items where product unavailable
        // TODO 2: Send notification to user about removed items
        // Queue email with list of removed items
```

```
        // TODO 3: Suggest alternative products if available
        // Query similar products by category and price range
        // TODO 4: Return updated cart summary with conflict resolution
    }

    async retryFailedOperation(operation, maxRetries = 3, baseDelay = 1000) {
        // TODO 1: Implement exponential backoff retry logic
        // Start with baseDelay, double on each retry
        // TODO 2: Distinguish between retryable and permanent failures
        // Network timeouts are retryable, validation errors are not
        // TODO 3: Log each retry attempt for monitoring
        // Include operation name, attempt number, and delay
        // TODO 4: Throw original error after max retries exceeded
    }
}
```

Language-Specific Error Handling Hints:

- **Database Transactions**: Use `db.beginTransaction()` and `await transaction.commit()` or `await transaction.rollback()`
- **Async Error Handling**: Always use try-catch blocks around await expressions, never rely on .catch() chaining alone
- **Express Error Middleware**: Error handlers must have 4 parameters: `(err, req, res, next)` - Express identifies error middleware by the parameter count
- **Promise Rejections**: Add `process.on('unhandledRejection')` handler to catch missed async errors
- **Timeout Implementation**: Use `Promise.race()` with `setTimeout()` to add timeouts to external service calls
- **Graceful Shutdown**: Handle SIGTERM and SIGINT signals to close database connections before process termination

Milestone Checkpoints:

After Milestone 1 (Product Catalog): Test error handling by temporarily breaking the database connection while loading the product listing page. The application should display a user-friendly message instead of crashing. Check that search functionality degrades gracefully when the search service is unavailable.

After Milestone 2 (Shopping Cart): Simulate inventory conflicts by having two browser windows attempt to add the last item of a product to their carts simultaneously. Verify that only one succeeds and the other receives an appropriate error message. Test cart persistence across browser restarts.

After Milestone 3 (User Authentication): Test authentication error scenarios including wrong passwords, expired sessions, and session store failures. Verify that users receive clear error messages and can recover by logging in again.

After Milestone 4 (Checkout Process): Test the complete error recovery flow by causing payment authorization to succeed but order creation to fail. Verify that the payment authorization gets voided and inventory gets released. Check that users can retry checkout after fixing any issues.

Debugging Tips:

Symptom	Likely Cause	Diagnosis Method	Fix
Orders created but payment never charged	Order creation succeeds but payment capture fails	Check payment gateway logs for authorization vs capture	Add payment capture step after order creation
Users report lost shopping carts	Session expiration or cleanup too aggressive	Check session store for expired sessions	Increase session timeout, implement cart persistence
Intermittent inventory oversells	Race conditions in inventory checking	Add logging around inventory queries, check for concurrent transactions	Implement pessimistic locking with SELECT FOR UPDATE
Payment authorizations not voided on failures	Missing compensating transaction in error handling	Search logs for payment authorization IDs without corresponding orders	Add payment void calls to all error handling paths
Error messages expose database details	Raw exceptions passed to users	Check error handler middleware for unhandled error types	Classify all errors as operational vs programming errors
Application crashes on external service failures	Missing circuit breaker protection	Monitor external service response times and error rates	Implement circuit breaker pattern with fallback logic

Testing Strategy

> **Milestone(s):** All milestones - outlines comprehensive testing approaches for product catalog (Milestone 1), shopping cart (Milestone 2), user authentication (Milestone 3), and checkout process (Milestone 4) with specific validation criteria and test scenarios

Testing an e-commerce platform requires a systematic approach that validates both individual components and their interactions across complex user workflows. Think of testing like quality control in a physical store - you need to verify that products display correctly, shopping carts function properly, checkout processes complete successfully, and the entire customer experience flows smoothly from browsing to purchase confirmation. The challenge lies in testing not just the happy path scenarios but also the numerous edge cases and failure conditions that can occur in a distributed system with concurrent users, inventory constraints, and external payment dependencies.

Testing Levels

Testing an e-commerce system involves multiple layers of validation, each targeting different aspects of functionality and integration. The testing pyramid guides our approach with unit tests forming the foundation, integration tests validating component interactions, and end-to-end tests ensuring complete user workflows function correctly.

Unit Testing focuses on individual functions and methods in isolation, using mocks and stubs to eliminate dependencies on external systems. This level catches logic errors, boundary conditions, and validates that each component behaves correctly when provided with specific inputs. Unit tests run quickly and provide immediate feedback during development, making them ideal for test-driven development practices.

Test Type	Target	Scope	Dependencies	Execution Speed	Feedback Quality
Unit	Individual functions/methods	Single component	Mocked	Very Fast (ms)	Precise error location
Integration	Component interactions	Multiple components	Real or test doubles	Medium (seconds)	Component boundary issues
End-to-End	Complete workflows	Full system	Real systems	Slow (minutes)	User experience validation

Integration Testing validates that components work correctly when combined, focusing on data flow between services, database interactions, and API contract compliance. These tests use real database connections but may mock external services like payment processors to maintain test reliability and speed.

End-to-End Testing exercises complete user workflows from browser interactions through backend processing, validating that the entire system delivers expected business outcomes. These tests run against a complete system deployment and catch issues that only emerge from the complex interactions between all components.

> **Critical Testing Principle**: Each testing level serves a distinct purpose in the validation strategy. Unit tests catch logic errors quickly during development. Integration tests validate component contracts and data flow. End-to-end tests ensure the complete user experience functions correctly. Skipping any level creates gaps in validation coverage that can allow bugs to reach production.

Component-Specific Testing Strategies target the unique challenges of each e-commerce component. The product catalog requires testing search performance with large datasets, pagination accuracy, and category hierarchy navigation. Shopping cart testing focuses on state management across sessions, inventory validation, and concurrent user operations. Authentication testing emphasizes security validation, session management, and password handling. Checkout testing validates complex transaction flows, inventory consistency, and failure recovery scenarios.

Component	Primary Test Focus	Key Challenges	Mock Requirements
Product Catalog	Search accuracy, pagination	Large datasets, performance	Image service, search index
Shopping Cart	State persistence, concurrency	Session management, timing	Inventory service, session store
Authentication	Security, session lifecycle	Timing attacks, token validation	Email service, session storage
Checkout	Transaction integrity	Race conditions, partial failures	Payment gateway, inventory system

Test Environment Strategy involves multiple isolated environments that mirror production configuration while providing controlled conditions for different testing scenarios. Development environments allow rapid iteration with synthetic data. Staging environments use production-like data volumes and external service integrations. Test environments provide controlled conditions for automated test execution without interfering with development work.

Milestone Checkpoints

Each milestone requires specific validation checkpoints that verify both functional requirements and technical implementation quality. These checkpoints provide concrete verification steps that confirm the system meets acceptance criteria before proceeding to the next milestone.

Milestone 1: Product Catalog Validation establishes the foundation for product discovery and browsing functionality. The validation process confirms that products display correctly, search functionality returns relevant results, pagination handles large

datasets appropriately, and category filtering narrows results accurately.

Checkpoint	Validation Method	Expected Behavior	Failure Indicators
Product Display	Load product listing page	Products show with images, prices, descriptions	Missing images, formatting errors, incorrect prices
Search Functionality	Enter search terms	Relevant products returned, ranked by relevance	No results for valid terms, irrelevant results
Pagination	Navigate through product pages	Consistent page sizes, accurate page counts	Duplicate products, missing items, incorrect totals
Category Filtering	Select category filters	Product list updates to show only matching items	Wrong products shown, filter state not maintained
Product Detail	Click individual product	Full product information displayed	Missing details, incorrect pricing, broken images

The product catalog validation process begins with database verification to ensure sample products are properly seeded with complete information including names, descriptions, prices, inventory quantities, and category assignments. The API endpoints must respond correctly to GET requests for product listings, individual products, and category hierarchies. Search functionality requires validation with various query types including exact matches, partial matches, and queries that should return no results.

Performance validation ensures the catalog handles realistic data volumes without degrading user experience. Test with datasets containing at least 1000 products across multiple categories to verify pagination performance and search response times remain acceptable. Load testing simulates concurrent users browsing the catalog to identify potential bottlenecks in database queries or search indexing.

Milestone 2: Shopping Cart Validation focuses on state management, item operations, and persistence across user sessions. The validation process confirms that cart operations maintain consistency, inventory validation prevents overselling, and cart state persists correctly across browser sessions and page refreshes.

Checkpoint	Validation Method	Expected Behavior	Failure Indicators
Add to Cart	Add products with various quantities	Items appear in cart with correct totals	Incorrect quantities, wrong prices, items not saved
Quantity Updates	Modify item quantities	Subtotals recalculate correctly	Math errors, negative quantities allowed, UI not updated
Item Removal	Remove items from cart	Items disappear, totals adjust	Items remain visible, incorrect totals
Cart Persistence	Refresh page, close/reopen browser	Cart contents maintained	Empty cart, lost items, session errors
Inventory Validation	Add more items than available	Error message, quantity limited to stock	Overselling allowed, no error feedback

Shopping cart validation begins with session management verification to ensure anonymous users receive consistent session identifiers and authenticated users maintain cart state across login/logout cycles. Item operations must handle edge cases including zero quantities, maximum quantity limits, and attempts to add discontinued products.

Concurrency testing simulates multiple users adding the same product to their carts simultaneously to verify inventory validation works correctly under race conditions. Session persistence testing validates cart state survives browser refresh, tab closure, and reasonable idle periods without losing customer selections.

Milestone 3: User Authentication Validation emphasizes security, session management, and user account operations. The validation process confirms that registration enforces security requirements, login authenticates credentials correctly, and session management maintains security while providing good user experience.

Checkpoint	Validation Method	Expected Behavior	Failure Indicators
User Registration	Submit registration form	Account created with hashed password	Plaintext passwords stored, weak validation
Login Authentication	Provide valid/invalid credentials	Correct authentication results	Wrong users logged in, timing differences
Password Security	Test password requirements	Strong passwords required, weak ones rejected	Weak passwords accepted, no complexity rules
Session Management	Login, navigate, idle, logout	Consistent authentication state	Sessions not expiring, state inconsistencies
Cart Transfer	Add items anonymously, then login	Anonymous cart merges with user account	Lost cart items, duplicate entries

Authentication validation starts with password security testing to ensure bcrypt hashing works correctly and password strength requirements prevent common weak passwords. Login timing testing verifies that authentication responses don't leak information about valid usernames through timing differences.

Session security testing validates that session tokens are cryptographically secure, expire appropriately, and resist common attacks like session fixation and CSRF. Cross-browser testing ensures authentication works consistently across different browsers and devices.

Milestone 4: Checkout Process Validation tests the complex transaction flows that convert shopping carts into confirmed orders. The validation process confirms that address collection works correctly, inventory validation prevents overselling, order creation maintains data consistency, and payment integration handles various scenarios appropriately.

Checkpoint	Validation Method	Expected Behavior	Failure Indicators
Address Collection	Submit shipping/billing forms	Valid addresses accepted, invalid rejected	Bad addresses accepted, good addresses rejected
Order Summary	Review order before confirmation	Accurate items, quantities, pricing, totals	Incorrect calculations, missing items
Inventory Validation	Checkout with limited inventory	Out-of-stock prevention, clear error messages	Orders exceeding stock, confusing errors
Order Creation	Complete checkout process	Order record created, cart cleared, confirmation shown	Partial orders, cart not cleared, no confirmation
Payment Integration	Submit payment information	Payment authorization simulation working	Payment errors not handled, status unclear

Checkout validation begins with address validation testing using various valid and invalid address formats to ensure the system correctly identifies deliverable addresses. Order processing testing validates that the atomic transaction correctly handles inventory updates, order record creation, and cart clearing without leaving the system in an inconsistent state.

Payment integration testing uses the stub payment service to simulate various payment scenarios including successful authorizations, declined payments, and network timeouts. Error handling testing validates that partial failures during checkout provide clear error messages and maintain system consistency.

Test Scenarios

Comprehensive test scenarios cover critical user paths, edge cases, and failure conditions that commonly occur in e-commerce systems. These scenarios validate both individual component functionality and complex interactions between components during realistic user workflows.

Happy Path Scenarios validate the most common user workflows that represent the majority of customer interactions. These scenarios establish baseline functionality and ensure the primary business value delivery works correctly.

Scenario	User Actions	Expected Outcome	Validation Points
Anonymous Browse and Purchase	Browse → Add to cart → Register → Checkout	Successful order creation	Cart transfer, address validation, payment
Returning Customer Purchase	Login → Browse → Add to cart → Checkout	Order using saved information	Session restoration, address autofill
Product Discovery	Search → Filter by category → Sort → View details	Relevant product found	Search accuracy, filter application
Cart Management	Add items → Update quantities → Remove item → Continue shopping	Accurate cart state	Persistence, calculation correctness

The anonymous browse and purchase scenario tests the complete customer journey from initial product discovery through order confirmation. This scenario validates that anonymous users can build shopping carts, register accounts during checkout, provide shipping addresses, and complete orders successfully. Key validation points include cart transfer from anonymous session to authenticated user, address validation during checkout, and payment authorization simulation.

Error Condition Scenarios test the system's behavior when operations fail or users encounter problems. These scenarios validate error handling, recovery mechanisms, and user experience during failure conditions.

Scenario	Trigger Condition	Expected Behavior	Recovery Validation
Inventory Conflict	Multiple users purchase last item	One succeeds, others get clear error	Cart updated with availability
Payment Failure	Payment authorization declined	Order not created, cart preserved	User can retry with different payment
Session Expiry	User idle beyond timeout	Login required for protected operations	Cart state preserved for recovery
System Unavailability	Database connection lost	Graceful error messages, retry options	Service restoration handling

Inventory conflict scenarios simulate race conditions where multiple users attempt to purchase the same limited-stock item simultaneously. The test validates that pessimistic locking prevents overselling while providing clear error messages to users who cannot complete their purchase due to insufficient inventory.

Security Test Scenarios validate that authentication, session management, and payment handling resist common attack vectors and maintain user data security.

Security Test	Attack Vector	Protection Mechanism	Validation Method
Password Brute Force	Rapid login attempts	Rate limiting, account lockout	Monitor failed attempt handling
Session Hijacking	Stolen session token	Secure cookies, HTTPS enforcement	Verify token security properties
SQL Injection	Malicious search queries	Parameterized queries	Submit injection payloads
CSRF Attack	Cross-site request forgery	CSRF tokens, origin validation	Attempt unauthorized operations

Password brute force testing validates that the system implements appropriate rate limiting to prevent automated password guessing attacks. The test submits rapid login attempts and verifies that the system temporarily locks accounts or implements exponential backoff delays to make brute force attacks impractical.

Performance Test Scenarios validate that the system maintains acceptable response times and handles concurrent user loads without degrading functionality or data consistency.

Performance Test	Load Condition	Performance Target	Degradation Indicators
Concurrent Browsing	100 simultaneous product searches	Response time < 2 seconds	Query timeouts, empty results
Cart Operations	50 users adding items simultaneously	No inventory overselling	Race condition failures
Authentication Load	25 simultaneous login attempts	All authentications complete	Failed logins for valid credentials
Checkout Concurrency	10 users checking out same products	Correct inventory handling	Overselling or system errors

Concurrent browsing tests simulate realistic user loads on the product catalog to identify database query bottlenecks and search performance issues. The test validates that pagination works correctly under load and that search results remain accurate when multiple users search simultaneously.

Data Consistency Scenarios test the system's ability to maintain accurate information across concurrent operations and failure conditions.

Consistency Test	Concurrent Operations	Expected Outcome	Inconsistency Indicators
Inventory Updates	Add to cart + Admin inventory change	Accurate stock levels	Negative inventory, overselling
Price Changes	Add to cart + Price update	Stable cart pricing	Cart totals change unexpectedly
Session Conflicts	Same user login from multiple devices	Latest session active	Multiple active sessions
Order Processing	Checkout + Inventory update	Atomic operations	Partial order completion

Inventory update tests validate that cart operations and administrative inventory changes maintain consistency when executed concurrently. The test simulates scenarios where administrators update product inventory while customers add items to their carts, ensuring that inventory levels remain accurate and overselling cannot occur.

Implementation Guidance

The testing strategy requires specific tools and frameworks that support the various testing levels while providing good developer experience and reliable test execution. The implementation focuses on practical testing approaches that junior developers can understand and maintain while ensuring comprehensive coverage of critical functionality.

Technology Recommendations for testing infrastructure provide simple options for getting started and advanced options for comprehensive testing as the system grows in complexity.

Testing Level	Simple Option	Advanced Option	Rationale
Unit Testing	Jest with Node.js built-ins	Jest + Sinon for complex mocking	Jest provides excellent developer experience
Integration Testing	Supertest for API testing	Supertest + Docker for isolated DB	Real database needed for integration validation
End-to-End	Playwright for browser automation	Playwright + CI/CD integration	Modern browser automation with good debugging
Database Testing	SQLite in-memory for speed	PostgreSQL test instance for realism	In-memory faster for unit tests
Mocking	Jest manual mocks	MSW for API mocking	Service worker mocking more realistic

Recommended Test File Structure organizes test files alongside source code while maintaining clear separation between different testing levels and concerns.

```
project-root/
├── src/
│   ├── components/
│   │   ├── product-catalog/
│   │   │   ├── product-service.js          ← business logic
│   │   │   ├── product-service.test.js     ← unit tests
│   │   │   ├── product-routes.js           ← API endpoints
│   │   │   └── product-routes.integration.test.js ← integration tests
│   │   ├── shopping-cart/
│   │   │   ├── cart-service.js
│   │   │   ├── cart-service.test.js
│   │   │   ├── cart-routes.js
│   │   │   └── cart-routes.integration.test.js
│   │   ├── authentication/
│   │   │   ├── auth-service.js
│   │   │   ├── auth-service.test.js
│   │   │   ├── password-service.js
│   │   │   └── password-service.test.js
│   │   └── checkout/
│   │       ├── checkout-service.js
│   │       ├── checkout-service.test.js
│   │       ├── order-service.js
│   │       └── order-service.test.js
├── tests/
│   ├── integration/
│   │   ├── api-workflow.test.js            ← cross-component API tests
│   │   ├── database-operations.test.js     ← database interaction tests
│   │   └── session-management.test.js      ← session persistence tests
│   ├── e2e/
│   │   ├── user-registration.test.js       ← complete user workflows
│   │   ├── product-discovery.test.js       ← browsing and search
│   │   ├── shopping-cart.test.js           ← cart management
│   │   └── checkout-process.test.js        ← order completion
│   ├── fixtures/
│   │   ├── sample-products.json            ← test data
│   │   ├── test-users.json                 ← user accounts
│   │   └── test-orders.json                ← order examples
│   └── helpers/
│       ├── database-setup.js               ← test DB initialization
│       ├── test-server.js                  ← test server setup
│       └── mock-services.js                ← external service mocks
└── package.json
```

Database Test Infrastructure provides isolated database instances for testing while ensuring tests run quickly and reliably without interfering with each other.

JAVASCRIPT

```javascript
// tests/helpers/database-setup.js

const sqlite3 = require('sqlite3');

const { open } = require('sqlite');

const { readFileSync } = require('fs');

const path = require('path');


class TestDatabase {

    constructor() {

        this.db = null;

    }


    async initialize() {

        // Create in-memory SQLite database for tests

        this.db = await open({

            filename: ':memory:',

            driver: sqlite3.Database

        });


        // Load schema from main application

        const schemaSQL = readFileSync(

            path.join(__dirname, '../../src/database/schema.sql'),

            'utf8'

        );

        await this.db.exec(schemaSQL);


        return this.db;

    }


    async seedTestData(fixture = 'basic') {

        const products = JSON.parse(

            readFileSync(path.join(__dirname, '../fixtures/sample-products.json'))

        );



        for (const product of products) {
```

```
            await this.db.run(`
                INSERT INTO products (name, description, price, inventory_quantity, category_id)
                VALUES (?, ?, ?, ?, ?)
            `, [product.name, product.description, product.price, product.inventory_quantity,
product.category_id]);
        }

        // Seed categories, users, etc.
        await this.seedCategories();
        await this.seedTestUsers();
    }

    async cleanup() {
        if (this.db) {
            await this.db.close();
            this.db = null;
        }
    }

    async seedCategories() {
        const categories = [
            { id: 1, name: 'Electronics', parent_id: null },
            { id: 2, name: 'Laptops', parent_id: 1 },
            { id: 3, name: 'Smartphones', parent_id: 1 },
            { id: 4, name: 'Books', parent_id: null }
        ];

        for (const category of categories) {
            await this.db.run(`
                INSERT INTO categories (id, name, parent_id, path)
                VALUES (?, ?, ?, ?)
            `, [category.id, category.name, category.parent_id, category.name]);
        }
    }
```

```
    async seedTestUsers() {
        const bcrypt = require('bcrypt');
        const passwordHash = await bcrypt.hash('testpassword123', 12);

        await this.db.run(`
            INSERT INTO users (email, password_hash)
            VALUES (?, ?)
        `, ['test@example.com', passwordHash]);
    }
}

module.exports = { TestDatabase };
```

API Testing Framework provides utilities for testing REST endpoints with proper setup, teardown, and assertion helpers that validate both success and error scenarios.

JAVASCRIPT

```javascript
// tests/helpers/test-server.js

const request = require('supertest');

const { EcommerceApp } = require('../../src/app');

const { TestDatabase } = require('./database-setup');


class TestServer {

    constructor() {

        this.app = null;

        this.database = null;

        this.server = null;

    }


    async start() {

        this.database = new TestDatabase();

        await this.database.initialize();

        await this.database.seedTestData();


        this.app = new EcommerceApp({

            database: this.database.db,

            sessionSecret: 'test-secret-key',

            environment: 'test'

        });


        await this.app.setupMiddleware();

        await this.app.setupRoutes();


        return this.app.express;

    }


    async stop() {

        if (this.database) {

            await this.database.cleanup();

        }

    }
```

```
    // Helper methods for common test operations
    async authenticateUser(email = 'test@example.com', password = 'testpassword123') {
        const response = await request(this.app.express)
            .post('/api/auth/login')
            .send({ email, password })
            .expect(200);

        return response.headers['set-cookie'];
    }

    async addProductToCart(sessionCookie, productId, quantity = 1) {
        return request(this.app.express)
            .post('/api/cart/add')
            .set('Cookie', sessionCookie)
            .send({ productId, quantity })
            .expect(200);
    }

    async createTestOrder(sessionCookie, shippingAddress = null) {
        const defaultAddress = {
            fullName: 'John Doe',
            addressLine1: '123 Main St',
            city: 'Springfield',
            state: 'IL',
            postalCode: '62701',
            country: 'US',
            phoneNumber: '555-123-4567'
        };

        return request(this.app.express)
            .post('/api/checkout/confirm')
            .set('Cookie', sessionCookie)
            .send({
                shippingAddress: shippingAddress || defaultAddress,
```

```
                paymentMethod: 'test-card'
            })
            .expect(200);
    }
}

module.exports = { TestServer };
```

Component Unit Test Skeleton provides the structure for testing individual service methods with proper mocking and assertion patterns.

JAVASCRIPT

```javascript
// src/components/product-catalog/product-service.test.js

const { ProductService } = require('./product-service');

const { TestDatabase } = require('../../tests/helpers/database-setup');


describe('ProductService', () => {

    let productService;

    let database;


    beforeEach(async () => {

        database = new TestDatabase();

        await database.initialize();

        await database.seedTestData();



        productService = new ProductService({ database: database.db });

    });


    afterEach(async () => {

        await database.cleanup();

    });


    describe('findWithFilters', () => {

        test('returns paginated product list with default options', async () => {

            // TODO: Call productService.findWithFilters with default options

            // TODO: Verify response contains ProductListResponse structure

            // TODO: Check pagination metadata is correct

            // TODO: Verify products array contains expected product fields

        });


        test('applies category filter correctly', async () => {

            // TODO: Filter by specific category ID

            // TODO: Verify all returned products belong to category

            // TODO: Check subcategory products are included

            // TODO: Validate pagination works with filtering

        });
```

```
        test('handles empty search results gracefully', async () => {
            // TODO: Search for non-existent product name
            // TODO: Verify empty products array returned
            // TODO: Check pagination shows zero total pages
            // TODO: Ensure no errors thrown for empty results
        });
    });

    describe('searchProducts', () => {
        test('finds products by exact name match', async () => {
            // TODO: Search for exact product name from test data
            // TODO: Verify matching product appears in results
            // TODO: Check search ranking puts exact match first
            // TODO: Validate SearchSuggestionResponse format
        });

        test('performs partial text search in descriptions', async () => {
            // TODO: Search using partial description text
            // TODO: Verify products with matching descriptions found
            // TODO: Check search handles case insensitivity
            // TODO: Validate relevance scoring works correctly
        });
    });

    describe('updateInventory', () => {
        test('decrements inventory for valid product', async () => {
            // TODO: Get initial inventory count for test product
            // TODO: Call updateInventory with negative quantity change
            // TODO: Verify inventory decreased by expected amount
            // TODO: Check product still accessible if inventory > 0
        });

        test('prevents inventory from going negative', async () => {
            // TODO: Attempt to decrement inventory below zero
```

```
            // TODO: Verify InventoryError thrown with appropriate details
            // TODO: Check inventory remains unchanged after error
            // TODO: Validate error message explains the conflict
        });
    });
});
```

Integration Test Framework validates component interactions and database operations with real database connections and HTTP requests.

JAVASCRIPT

```javascript
// tests/integration/shopping-cart-api.test.js

const { TestServer } = require('../helpers/test-server');

const request = require('supertest');


describe('Shopping Cart API Integration', () => {

    let testServer;

    let app;


    beforeAll(async () => {

        testServer = new TestServer();

        app = await testServer.start();

    });


    afterAll(async () => {

        await testServer.stop();

    });


    describe('Anonymous Cart Operations', () => {

        test('creates cart session and adds item', async () => {

            // TODO: Send POST to /api/cart/add without authentication

            // TODO: Verify session cookie set in response headers

            // TODO: Check cart item added with correct quantity

            // TODO: Validate cart summary includes pricing information

        });


        test('persists cart across multiple requests', async () => {

            // TODO: Add item to cart and capture session cookie

            // TODO: Make subsequent request using same cookie

            // TODO: Verify cart contents preserved between requests

            // TODO: Check cart expiration time updates with activity

        });

    });


    describe('Authenticated Cart Operations', () => {

        test('transfers anonymous cart on user login', async () => {
```

```
            // TODO: Add items to anonymous cart

            // TODO: Login with test user credentials

            // TODO: Verify anonymous cart items now associated with user

            // TODO: Check session transitioned from anonymous to authenticated

        });


        test('validates inventory on cart operations', async () => {

            // TODO: Add maximum available quantity to cart

            // TODO: Attempt to add more of same product

            // TODO: Verify InventoryError response with current availability

            // TODO: Check cart quantity limited to available stock

        });

    });

});
```

End-to-End Test Framework uses browser automation to validate complete user workflows from frontend interactions through backend processing.

JAVASCRIPT

```javascript
// tests/e2e/checkout-process.test.js

const { test, expect } = require('@playwright/test');


test.describe('Checkout Process', () => {

    test.beforeEach(async ({ page }) => {

        await page.goto('http://localhost:3000');

    });


    test('completes anonymous user purchase flow', async ({ page }) => {

        // TODO: Browse to product catalog page

        // TODO: Search for specific product

        // TODO: Add product to cart with specified quantity

        // TODO: Navigate to cart and verify contents

        // TODO: Proceed to checkout and create account

        // TODO: Fill shipping address form

        // TODO: Complete payment with test card

        // TODO: Verify order confirmation page displays

        // Validation points at each step

    });


    test('handles inventory conflicts during checkout', async ({ page }) => {

        // TODO: Add last available item to cart

        // TODO: Simulate another user purchasing same item

        // TODO: Attempt to complete checkout

        // TODO: Verify inventory conflict error message

        // TODO: Check cart updated with current availability

        // TODO: Validate user can modify cart and retry

    });


    test('preserves cart during account creation', async ({ page }) => {

        // TODO: Add multiple items to cart as anonymous user

        // TODO: Proceed to checkout and trigger registration

        // TODO: Complete registration form

        // TODO: Verify cart contents preserved after account creation
```

```
        // TODO: Complete checkout with new account
        // TODO: Check order confirmation includes original cart items
    });
});
```

Milestone Validation Checkpoints provide specific commands and expected outcomes for verifying each milestone meets acceptance criteria.

Milestone 1 Checkpoint - Product Catalog:

```bash
# Run unit tests for catalog component
npm test -- src/components/product-catalog/

# Run integration tests for catalog API
npm test -- tests/integration/product-catalog-api.test.js

# Start development server for manual testing
npm run dev

# Test manual workflows:
# 1. Visit http://localhost:3000/products
# 2. Verify products display with images and prices
# 3. Test search functionality with various terms
# 4. Navigate through pagination
# 5. Apply category filters and verify results
```

Milestone 2 Checkpoint - Shopping Cart:

```bash
# Test cart functionality
npm test -- src/components/shopping-cart/
npm test -- tests/integration/shopping-cart-api.test.js

# Manual validation:
# 1. Add items to cart and verify persistence
# 2. Update quantities and check calculations
# 3. Test cart across browser refresh
# 4. Try to add more items than available inventory
```

Milestone 3 Checkpoint - User Authentication:

```bash
# Test authentication security
npm test -- src/components/authentication/
npm test -- tests/integration/authentication-flow.test.js

# Security validation:
# 1. Register new account with weak password (should fail)
# 2. Login with correct and incorrect credentials
# 3. Verify session persists across browser refresh
# 4. Test session expiration after idle timeout
```

Milestone 4 Checkpoint - Checkout Process:

```bash
# Test complete checkout flow
npm test -- src/components/checkout/
npm test -- tests/integration/checkout-process.test.js
npm test -- tests/e2e/

# End-to-end validation:
# 1. Complete purchase as anonymous user
# 2. Test checkout with inventory conflicts
# 3. Verify order confirmation and inventory updates
# 4. Test payment failure handling
```

Performance Testing Commands validate system behavior under load:

```bash
# Install load testing tool
npm install -g artillery

# Run performance tests
artillery run tests/performance/catalog-load-test.yml
artillery run tests/performance/cart-concurrency-test.yml

# Expected results:
# - Response times under 2 seconds for catalog queries
# - No inventory overselling under concurrent cart operations
# - Session management stable under authentication load
```

Debugging Guide

> **Milestone(s):** All milestones - provides symptom-cause-fix tables and debugging techniques that apply across product catalog (Milestone 1), shopping cart (Milestone 2), user authentication (Milestone 3), and checkout process (Milestone 4) components

Debugging an e-commerce system requires understanding how multiple components interact and where failures commonly occur. Think of debugging like being a detective investigating a crime scene - you need to gather evidence, follow the trail of clues, and reconstruct what happened step by step. Unlike debugging a simple script, e-commerce systems have multiple moving parts: databases maintaining inventory, sessions tracking user state, authentication systems managing security, and checkout processes coordinating atomic transactions. A single user action like "add to cart" touches session management, inventory validation, cart persistence, and price calculations. When something goes wrong, the challenge is isolating which component failed and why.

The debugging approach for e-commerce systems follows a structured methodology. First, reproduce the issue reliably to understand the failure pattern. Second, examine the application state at the point of failure through logs, database queries, and session inspection. Third, trace the request flow backwards from the error to identify where the system deviated from expected behavior. Fourth, validate your hypothesis by making targeted changes and observing the results. This systematic approach prevents the common mistake of making random changes without understanding root cause.

E-commerce debugging is particularly challenging because of the stateful nature of user interactions. Unlike stateless operations that can be debugged in isolation, cart management, authentication, and checkout processes maintain state across multiple requests. A bug might manifest during checkout but originate from a cart operation performed hours earlier. Session expiration, inventory changes, and price updates can all cause delayed failures that appear unrelated to their root cause.

Common Bug Patterns

Understanding typical failure patterns helps developers quickly identify and resolve issues. Each milestone introduces specific categories of bugs that manifest in predictable ways. The key is recognizing the symptoms and knowing which debugging techniques apply to each situation.

Product Catalog Issues

Product catalog problems typically manifest as incorrect search results, missing products, or performance issues during browsing. These bugs often stem from database query problems, caching issues, or incorrect filtering logic.

Bug Pattern	Symptoms	Root Cause	Diagnosis Steps	Fix Strategy
Missing Products in Search	Search returns incomplete results, pagination shows wrong counts	Database query excludes products incorrectly, N+1 query problems	Check SQL query with EXPLAIN, examine WHERE clauses	Fix query logic, add proper indexes, optimize joins
Slow Product Loading	Page load times exceed 5 seconds, timeouts during browsing	Missing database indexes, inefficient joins, large image loading	Use query profiler, check slow query logs, monitor network requests	Add indexes on frequently queried columns, implement image lazy loading
Incorrect Price Display	Prices show wrong values, currency formatting issues	Floating-point precision errors, currency conversion problems	Check price calculations in debugger, examine database storage	Use integer cents for storage, implement proper currency formatting
Category Hierarchy Broken	Products appear in wrong categories, navigation fails	Recursive query problems, corrupted parent_id references	Examine category table structure, test recursive queries	Fix referential integrity, implement cycle detection
Search Ranking Wrong	Relevant products appear last, poor search experience	Scoring algorithm issues, missing text indexes	Test search queries manually, examine ranking calculations	Implement proper full-text search, tune ranking weights

The most common product catalog bug is the N+1 query problem when displaying product lists. Developers implement the basic product retrieval correctly but forget to eager-load related data like categories and images. This results in one query to get products, then N additional queries to fetch category information for each product. The symptom is dramatically slower page loads as the product count increases.

⚠️ **Pitfall: Floating-Point Price Storage** Many developers store prices as floating-point numbers, leading to precision errors during calculations. For example, storing $19.99 as a float might result in 19.989999999... causing incorrect totals during cart calculations. Always store monetary values as integers representing cents, then convert to decimal format for display only.

Shopping Cart State Management Issues

Shopping cart bugs are particularly tricky because they involve session management, inventory validation, and state synchronization across multiple requests. Cart state can become inconsistent when concurrent operations conflict or when session storage fails.

Bug Pattern	Symptoms	Root Cause	Diagnosis Steps	Fix Strategy
Cart Items Disappear	Items vanish between page loads, cart shows empty unexpectedly	Session expiration, storage conflicts, improper cart transfer	Check session storage, examine cart transfer logs, verify session timeouts	Implement proper session renewal, fix cart transfer logic
Quantity Updates Ignored	Cart shows old quantities after updates, changes don't persist	Race conditions, validation failures, session conflicts	Test concurrent updates, check validation logic, examine database locks	Add optimistic locking, implement proper validation, use atomic updates
Price Mismatch at Checkout	Cart total differs from checkout total, old prices displayed	Price staleness, cache invalidation issues, concurrent price changes	Compare cart timestamps with price update logs, check cache behavior	Implement price validation at checkout, add cache invalidation
Inventory Over-Allocation	Cart allows more items than available stock, checkout fails	Inadequate inventory checks, race conditions during add-to-cart	Test concurrent add operations, examine inventory locking	Use pessimistic locking for inventory, validate at each step
Session Transfer Failure	Cart disappears after login, anonymous cart not merged	Authentication flow issues, incorrect cart association	Trace authentication flow, check cart ownership logic	Fix cart transfer logic, handle edge cases properly

Cart debugging requires understanding the cart state machine and how state transitions occur. The most complex bugs involve race conditions when multiple browser tabs perform cart operations simultaneously. For example, if a user opens two tabs and adds different items to the cart, the system might lose one of the additions due to improper state synchronization.

⚠️ **Pitfall: Client-Side Cart State** Some developers maintain cart state entirely in browser localStorage or cookies, leading to synchronization issues and security vulnerabilities. Cart state should always be authoritative on the server side, with client-side state used only for immediate UI updates before server confirmation.

Authentication and Session Management Issues

Authentication bugs can range from security vulnerabilities to user experience problems. These issues often involve session handling, password validation, or token management problems.

Bug Pattern	Symptoms	Root Cause	Diagnosis Steps	Fix Strategy
Session Expiry During Checkout	Users logged out mid-checkout, session timeout errors	Inappropriate session timeout, missing activity renewal	Check session timeout configuration, trace activity updates	Implement sliding session expiration, add checkout session extension
Password Hash Failures	Login works inconsistently, password verification fails randomly	Incorrect bcrypt implementation, salt handling issues	Test password hashing with known values, check bcrypt parameters	Fix bcrypt usage, ensure consistent salt handling
Session Fixation Vulnerability	User sessions persist after logout, security scanner alerts	Improper session destruction, session ID reuse	Test session lifecycle, check session cleanup logic	Implement proper session invalidation, regenerate session IDs
Authentication Bypass	Protected routes accessible without login, middleware failures	Middleware ordering issues, incomplete route protection	Test protected endpoints without authentication, check middleware application	Fix middleware order, ensure comprehensive route protection
Timing Attack Vulnerability	Login response times vary based on user existence	User lookup implementation reveals information	Measure response times for valid/invalid users	Implement constant-time comparison, add artificial delays

Authentication debugging requires testing both positive and negative cases. Many developers test successful login flows but neglect to verify that invalid attempts are handled securely. Timing attacks are particularly subtle - the system appears to work correctly but reveals information through response time patterns.

⚠️ **Pitfall: Weak Password Hashing** Using inadequate hashing algorithms like MD5 or SHA-1, or implementing bcrypt with too few rounds, creates security vulnerabilities. Always use bcrypt with at least 12 salt rounds, and verify the implementation by testing with known password/hash pairs.

Checkout Process Transaction Issues

Checkout bugs are the most critical because they directly affect revenue and customer experience. These issues typically involve transaction isolation, inventory consistency, or payment processing failures.

Bug Pattern	Symptoms	Root Cause	Diagnosis Steps	Fix Strategy
Inventory Double-Booking	Multiple orders for same item exceed available stock	Race conditions, inadequate locking, poor transaction isolation	Test concurrent checkouts, examine database locks, check transaction logs	Implement SELECT FOR UPDATE, use proper transaction isolation
Partial Order Creation	Orders created with missing items, incomplete data	Transaction rollback failures, exception handling issues	Examine transaction boundaries, check rollback logic	Ensure atomic order creation, implement proper error handling
Payment Authorization Leak	Payment authorized but order creation fails, funds held unnecessarily	Transaction coordination issues, payment system integration problems	Trace payment/order creation sequence, check authorization handling	Implement compensation logic, add payment void on order failure
Address Validation Failures	Orders created with invalid addresses, shipping problems	Inadequate address validation, geocoding service issues	Test address validation edge cases, check external service integration	Implement robust address validation, add fallback mechanisms
Checkout Session Corruption	Checkout process fails with data inconsistency errors	Session data corruption, concurrent modification	Examine checkout session lifecycle, test concurrent access	Add checkout session isolation, implement integrity checks

Checkout debugging requires understanding transaction boundaries and ACID properties. The most dangerous bugs involve partial state updates where some operations succeed while others fail, leaving the system in an inconsistent state.

⚠️ **Pitfall: Insufficient Transaction Scope** Many developers create transactions that are too narrow, leading to consistency issues. For example, creating an order record in one transaction and updating inventory in another allows race conditions where inventory becomes negative. Always encompass related operations within a single transaction boundary.

Debugging Techniques

Effective debugging requires systematic data collection and analysis. The key is gathering sufficient information to understand system behavior without overwhelming yourself with irrelevant details.

Structured Logging Strategy

Implementing comprehensive logging provides the foundation for effective debugging. E-commerce applications need logs that capture both user actions and system state changes.

The logging strategy follows a hierarchical approach. At the request level, log the user session, requested operation, and input parameters. At the business logic level, log state transitions, validation results, and external service calls. At the data access level, log database operations, query performance, and transaction boundaries.

Log Level	Content	Example Use Case
ERROR	System failures, unhandled exceptions	Payment processing failures, database connection errors
WARN	Recoverable issues, degraded performance	Inventory conflicts resolved, session near expiration
INFO	Business operations, state changes	User login, cart modifications, order creation
DEBUG	Detailed execution flow, variable values	Function entry/exit, intermediate calculations
TRACE	Fine-grained execution details	Database query parameters, API request/response bodies

Each log entry should include contextual information that enables correlation across system components. The standard log format includes timestamp, log level, component name, session identifier, user identifier (if authenticated), and message content.

> **Key Insight**: Correlation identifiers are essential for tracing requests across components. Generate a unique request ID when the request enters the system and include it in all related log entries. This allows you to follow a single user's journey through multiple system components.

State Inspection Methods

E-commerce debugging requires examining both persistent state (database) and transient state (sessions, caches). The inspection strategy depends on the type of failure and affected components.

Database state inspection involves examining table contents, foreign key relationships, and constraint violations. For product catalog issues, check product visibility flags, category hierarchies, and inventory quantities. For cart problems, examine cart and cart item records, looking for orphaned entries or invalid foreign keys. For authentication issues, verify user records, session storage, and password hashes.

Session state inspection requires accessing session storage and examining session data structure. Most session-related bugs involve expired sessions, corrupted session data, or improper session transfer between anonymous and authenticated states.

State Type	Inspection Method	Tools	Key Information
Database	Direct SQL queries	Database client, query profiler	Record existence, foreign key validity, constraint violations
Session Storage	Session debugging endpoints	Browser dev tools, server logs	Session expiration, data completeness, ownership
Application Cache	Cache inspection utilities	Redis CLI, Memcached stats	Cache hit rates, data freshness, eviction patterns
Request Context	Middleware logging	Application logs, debugging middleware	Authentication state, request parameters, response codes

Request Flow Tracing

Understanding how requests flow through the system helps identify where failures occur. The tracing approach follows requests from entry point through all system layers to the final response.

Request flow tracing begins with HTTP request logging at the web server level. Capture the request method, URL, headers, and body content. Follow the request through routing logic, middleware execution, and business logic processing. Track database queries, external service calls, and response generation.

The systematic tracing process involves five steps. First, identify the request entry point and initial parameters. Second, trace the request through each middleware component, noting any modifications to request context. Third, follow the request into business logic, examining validation steps and data transformations. Fourth, track database operations and external service calls. Fifth, trace response generation and any cleanup operations.

For complex operations like checkout, create a request flow diagram showing decision points, external service calls, and state modifications. This visual representation helps identify potential failure points and race condition windows.

Performance Debugging Techniques

Performance issues in e-commerce systems often stem from database inefficiencies, caching problems, or resource contention. The debugging approach focuses on identifying bottlenecks and measuring performance impact.

Database performance debugging starts with query analysis. Use database query profilers to identify slow queries, missing indexes, and inefficient joins. The EXPLAIN command reveals query execution plans and helps optimize database access patterns.

Application performance debugging involves profiling CPU usage, memory consumption, and I/O operations. For Node.js applications, use built-in profiling tools or external profilers to identify performance hotspots.

Performance Issue	Detection Method	Analysis Approach	Resolution Strategy
Slow Database Queries	Query profiler, slow query logs	EXPLAIN analysis, index usage review	Add indexes, optimize queries, implement caching
Memory Leaks	Memory monitoring, heap snapshots	Object retention analysis	Fix object references, implement proper cleanup
CPU Bottlenecks	CPU profiling, performance monitoring	Function-level profiling	Optimize algorithms, implement caching, scale horizontally
I/O Contention	System monitoring, disk usage analysis	I/O pattern analysis	Optimize file access, implement connection pooling

Troubleshooting Tools

Effective debugging requires the right tools for different types of problems. The tool selection depends on the failure symptom and affected system components.

Development Environment Tools

The development environment provides immediate feedback and detailed debugging capabilities. These tools help identify issues during development before they reach production.

Browser developer tools are essential for client-side debugging. The Network tab reveals API request/response details, timing information, and error responses. The Application tab shows localStorage, sessionStorage, and cookie contents. The Console tab displays JavaScript errors and custom debug output.

Database management tools provide direct access to database contents and query execution. Use database clients to examine table contents, test queries, and verify data integrity. Query profilers help identify performance issues and optimization opportunities.

Tool Category	Specific Tools	Primary Use Cases
Browser DevTools	Chrome/Firefox DevTools	API debugging, session inspection, performance analysis
Database Clients	SQLite Browser, pgAdmin, MySQL Workbench	Data verification, query testing, schema inspection
API Testing	Postman, curl, Insomnia	Endpoint testing, authentication verification, response validation
Code Debugging	VS Code debugger, Node.js inspector	Breakpoint debugging, variable inspection, call stack analysis
Log Analysis	grep, awk, log viewers	Pattern matching, error correlation, performance analysis

Production Monitoring Tools

Production debugging requires non-intrusive monitoring tools that provide system visibility without affecting performance. These tools help identify issues in live environments where interactive debugging isn't feasible.

Application monitoring tools track request rates, response times, error rates, and resource utilization. They provide dashboards for system health and alerting for anomalous behavior. Database monitoring tools track query performance, connection usage, and lock contention.

Log aggregation tools centralize log data from multiple system components, enabling correlation and pattern analysis. They support filtering, searching, and real-time monitoring of log streams.

Monitoring Aspect	Tools	Key Metrics
Application Performance	New Relic, DataDog, AppDynamics	Response time, throughput, error rate, resource utilization
Database Performance	pg_stat_statements, MySQL Performance Schema	Query time, lock waits, connection pool usage
Infrastructure	Prometheus, Grafana, CloudWatch	CPU, memory, disk I/O, network traffic
Log Aggregation	ELK Stack, Splunk, Fluentd	Log volume, error patterns, user behavior

Debugging Workflow Integration

Integrating debugging tools into the development workflow ensures consistent debugging practices and efficient problem resolution. The integration approach combines automated monitoring with manual investigation techniques.

The debugging workflow begins with issue detection through monitoring alerts or user reports. The initial triage involves classifying the issue severity and affected system components. For critical issues affecting checkout or authentication, implement immediate workarounds while investigating root cause.

The investigation phase uses appropriate debugging tools based on the issue type. For database issues, use query profilers and database monitoring tools. For authentication problems, examine session storage and authentication logs. For performance issues, use application profilers and system monitoring tools.

Documentation of debugging sessions helps build organizational knowledge and prevents recurring issues. Record the issue symptoms, investigation steps, root cause analysis, and resolution strategy. This documentation becomes valuable reference material for future debugging efforts.

> **Architecture Decision: Centralized Logging Strategy**
>
> - **Context**: Multiple system components generate logs that need correlation for effective debugging
> - **Options Considered**: Component-specific log files, centralized logging service, hybrid approach
> - **Decision**: Implement centralized logging with structured format and correlation identifiers
> - **Rationale**: Centralized logging enables request tracing across components and simplifies debugging workflow
> - **Consequences**: Requires additional infrastructure but provides superior debugging capabilities and system visibility

The debugging process follows a structured escalation path. Level 1 debugging uses browser developer tools and application logs to identify obvious issues. Level 2 debugging involves database inspection, session analysis, and code-level investigation. Level 3 debugging requires production monitoring data, performance profiling, and deep system analysis.

Implementation Guidance

This section provides practical tools and techniques for implementing debugging capabilities in your e-commerce system. The focus is on building debugging infrastructure that supports efficient problem resolution during development and production operation.

Technology Recommendations

Component	Simple Option	Advanced Option
Logging Framework	console.log with timestamp formatting	Winston or Pino with structured logging
Database Debugging	SQLite Browser with manual queries	Database query profiler with automated analysis
API Testing	curl commands with JSON formatting	Postman collections with automated testing
Performance Monitoring	Node.js built-in profiler	Application Performance Monitoring (APM) service
Error Tracking	Try-catch blocks with console output	Sentry or Bugsnag for centralized error reporting
Session Debugging	Manual session storage inspection	Express session debugging middleware

Recommended File Structure

Organize debugging utilities and monitoring code into dedicated modules to keep debugging concerns separate from business logic:

```
project-root/
├── src/
│   ├── debug/
│   │   ├── logger.js              ← centralized logging configuration
│   │   ├── session-inspector.js   ← session debugging utilities
│   │   ├── db-profiler.js         ← database query profiling
│   │   └── error-handler.js       ← standardized error handling
│   ├── middleware/
│   │   ├── request-logger.js      ← HTTP request/response logging
│   │   ├── error-middleware.js    ← Express error handling middleware
│   │   └── debug-middleware.js    ← development debugging helpers
│   ├── utils/
│   │   ├── validation-helpers.js  ← input validation utilities
│   │   └── test-helpers.js        ← testing utility functions
│   └── routes/
│       └── debug-routes.js        ← debugging endpoints (dev only)
├── logs/
│   ├── application.log            ← structured application logs
│   ├── error.log                  ← error-specific logs
│   └── debug.log                  ← detailed debugging information
└── scripts/
    ├── analyze-logs.js            ← log analysis utilities
    └── db-health-check.js         ← database diagnostic script
```

Logging Infrastructure Starter Code

JAVASCRIPT

```
// src/debug/logger.js

const winston = require('winston');

const path = require('path');


// Create logger configuration with multiple transport options

const logger = winston.createLogger({

  level: process.env.LOG_LEVEL || 'info',

  format: winston.format.combine(

    winston.format.timestamp(),

    winston.format.errors({ stack: true }),

    winston.format.json(),

    winston.format.printf(({ timestamp, level, message, ...meta }) => {

      const correlationId = meta.correlationId || 'unknown';

      const userId = meta.userId || 'anonymous';

      const sessionId = meta.sessionId || 'no-session';



      return JSON.stringify({

        timestamp,

        level,

        message,

        correlationId,

        userId,

        sessionId,

        ...meta

      });

    })

  ),

  transports: [

    new winston.transports.File({

      filename: path.join(__dirname, '../../logs/error.log'),

      level: 'error'
```

```
    }),
    new winston.transports.File({
      filename: path.join(__dirname, '../../logs/application.log')
    })
  ]
});

// Add console output for development environment
if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    format: winston.format.combine(
      winston.format.colorize(),
      winston.format.simple()
    )
  }));
}

// Helper function to create contextual logger with correlation ID
function createContextLogger(correlationId, userId = null, sessionId = null) {
  return {
    error: (message, meta = {}) => logger.error(message, { correlationId, userId, sessionId, ...meta }),
    warn: (message, meta = {}) => logger.warn(message, { correlationId, userId, sessionId, ...meta }),
    info: (message, meta = {}) => logger.info(message, { correlationId, userId, sessionId, ...meta }),
    debug: (message, meta = {}) => logger.debug(message, { correlationId, userId, sessionId, ...meta })
  };
}

module.exports = { logger, createContextLogger };
```

JAVASCRIPT

```javascript
// src/middleware/request-logger.js

const { v4: uuidv4 } = require('uuid');

const { createContextLogger } = require('../debug/logger');


// Middleware to log all HTTP requests with correlation tracking

function requestLogger(req, res, next) {

  // Generate unique correlation ID for request tracing

  const correlationId = uuidv4();

  req.correlationId = correlationId;



  // Extract user context from session

  const userId = req.session?.userId || null;

  const sessionId = req.session?.id || null;



  // Create contextual logger for this request

  req.logger = createContextLogger(correlationId, userId, sessionId);



  // Log incoming request details

  req.logger.info('Incoming request', {

    method: req.method,

    url: req.url,

    userAgent: req.get('User-Agent'),

    ip: req.ip,

    body: req.method === 'POST' ? req.body : undefined

  });



  // Capture response details

  const originalSend = res.send;

  res.send = function(data) {

    req.logger.info('Outgoing response', {

      statusCode: res.statusCode,

      responseSize: Buffer.byteLength(data),
```

```
      duration: Date.now() - req.startTime
    });
    originalSend.call(this, data);
  };

  req.startTime = Date.now();
  next();
}

module.exports = requestLogger;
```

Database Debugging Utilities

JAVASCRIPT

```
// src/debug/db-profiler.js

const { logger } = require('./logger');


class DatabaseProfiler {

  constructor(database) {

    this.db = database;

    this.queryLog = [];

    this.slowQueryThreshold = 100; // milliseconds

  }


  // Wrap database query with performance monitoring

  async profileQuery(sql, params = [], correlationId = 'unknown') {

    const startTime = Date.now();

    const queryId = `${correlationId}-${Date.now()}`;



    try {

      logger.debug('Executing database query', {

        queryId,

        sql: sql.replace(/\s+/g, ' ').trim(),

        params,

        correlationId

      });



      const result = await this.db.all(sql, params);

      const duration = Date.now() - startTime;



      // Log slow queries for optimization

      if (duration > this.slowQueryThreshold) {

        logger.warn('Slow database query detected', {

          queryId,

          sql,

          duration,
```

```
        resultCount: result.length,
        correlationId
      });
    }

    // Track query for analysis
    this.queryLog.push({
      queryId,
      sql,
      params,
      duration,
      resultCount: result.length,
      timestamp: new Date()
    });

    return result;
  } catch (error) {
    const duration = Date.now() - startTime;
    logger.error('Database query failed', {
      queryId,
      sql,
      params,
      error: error.message,
      duration,
      correlationId
    });
    throw error;
  }
}

// Analyze query patterns for optimization opportunities
getQueryAnalysis() {
```

```
    const totalQueries = this.queryLog.length;
    const slowQueries = this.queryLog.filter(q => q.duration > this.slowQueryThreshold);
    const averageQueryTime = this.queryLog.reduce((sum, q) => sum + q.duration, 0) / totalQueries;

    return {
      totalQueries,
      slowQueryCount: slowQueries.length,
      averageQueryTime,
      slowestQuery: this.queryLog.sort((a, b) => b.duration - a.duration)[0],
      queryPatterns: this.groupQueriesByPattern()
    };
  }

  groupQueriesByPattern() {
    const patterns = {};
    this.queryLog.forEach(query => {
      // Normalize SQL by removing parameter values
      const pattern = query.sql.replace(/\$\d+|\?/g, '?').replace(/\s+/g, ' ').trim();
      if (!patterns[pattern]) {
        patterns[pattern] = { count: 0, totalTime: 0, examples: [] };
      }
      patterns[pattern].count++;
      patterns[pattern].totalTime += query.duration;
      if (patterns[pattern].examples.length < 3) {
        patterns[pattern].examples.push(query);
      }
    });

    return patterns;
  }
}
```

```
module.exports = DatabaseProfiler;
```

Error Handling Infrastructure

JAVASCRIPT

```
// src/debug/error-handler.js

const { logger } = require('./logger');


class AppError extends Error {

  constructor(message, statusCode = 500, code = 'INTERNAL_ERROR', details = {}) {

    super(message);

    this.statusCode = statusCode;

    this.code = code;

    this.details = details;

    this.timestamp = new Date();

    this.operational = true; // Marks as operational error vs programming error



    Error.captureStackTrace(this, this.constructor);

  }

}


class ValidationError extends AppError {

  constructor(field, value, constraint) {

    super(`Validation failed for field '${field}': ${constraint}`, 400, 'VALIDATION_ERROR', {

      field,

      value,

      constraint

    });

  }

}


class InventoryError extends AppError {

  constructor(productId, requested, available) {

    super(`Insufficient inventory for product ${productId}`, 409, 'INVENTORY_UNAVAILABLE', {

      productId,

      requested,

      available
```

```
    });
  }
}

class PaymentError extends AppError {
  constructor(message, paymentDetails = {}) {
    super(message, 402, 'PAYMENT_FAILED', paymentDetails);
  }
}

// Centralized error handling for Express applications
function errorHandler(err, req, res, next) {
  const correlationId = req.correlationId || 'unknown';
  const userId = req.session?.userId || null;
  const sessionId = req.session?.id || null;

  // Log error details for debugging
  logger.error('Request error occurred', {
    correlationId,
    userId,
    sessionId,
    error: err.message,
    stack: err.stack,
    statusCode: err.statusCode || 500,
    code: err.code || 'UNKNOWN_ERROR',
    details: err.details || {},
    url: req.url,
    method: req.method
  });

  // Send appropriate response based on error type
  if (err instanceof ValidationError) {
    return res.status(err.statusCode).json({
```

```
    success: false,
    error: {
      code: err.code,
      message: err.message,
      field: err.details.field,
      value: err.details.value
    },
    correlationId
  });
}

if (err instanceof InventoryError) {
  return res.status(err.statusCode).json({
    success: false,
    error: {
      code: err.code,
      message: err.message,
      productId: err.details.productId,
      available: err.details.available
    },
    correlationId
  });
}

// Generic error response (avoid exposing internal details)
const statusCode = err.statusCode || 500;
res.status(statusCode).json({
  success: false,
  error: {
    code: err.code || 'INTERNAL_ERROR',
    message: statusCode === 500 ? 'Internal server error' : err.message
  },
```

```
    correlationId
  });
}

module.exports = {
  AppError,
  ValidationError,
  InventoryError,
  PaymentError,
  errorHandler
};
```

Session Debugging Utilities

```javascript
// src/debug/session-inspector.js

const { createContextLogger } = require('./logger');


class SessionInspector {

  constructor(sessionStore) {

    this.sessionStore = sessionStore;

  }


  // TODO 1: Implement session data retrieval with error handling

  // TODO 2: Add session validation checking for required fields

  // TODO 3: Include session expiration analysis

  // TODO 4: Provide session transfer debugging information

  async inspectSession(sessionId, correlationId = 'debug') {

    // Implementation needed: retrieve session data and validate structure

  }


  // TODO 1: Query session store for all sessions matching criteria

  // TODO 2: Filter sessions by user ID, creation time, or expiration status

  // TODO 3: Return summary statistics for session analysis

  async findSessions(criteria = {}) {

    // Implementation needed: search and filter session records

  }


  // TODO 1: Check session expiration against current time

  // TODO 2: Validate required session fields (userId, cartId, etc.)

  // TODO 3: Verify session data integrity and format

  validateSession(sessionData) {

    // Implementation needed: comprehensive session validation

  }


  // TODO 1: Compare anonymous and authenticated session data

  // TODO 2: Identify cart transfer success/failure patterns

  // TODO 3: Report on data migration completeness
```

```
  debugCartTransfer(anonymousSessionId, authenticatedSessionId) {
    // Implementation needed: cart transfer analysis
  }
}

module.exports = SessionInspector;
```

Development Debugging Endpoints

JAVASCRIPT

```javascript
// src/routes/debug-routes.js - Only enabled in development environment
const express = require('express');
const { SessionInspector } = require('../debug/session-inspector');
const { DatabaseProfiler } = require('../debug/db-profiler');

const router = express.Router();

// Only enable debug routes in development
if (process.env.NODE_ENV !== 'production') {


  // Endpoint to inspect current session state
  router.get('/debug/session', async (req, res) => {
    try {
      const sessionId = req.session.id;
      const inspector = new SessionInspector(req.sessionStore);
      const sessionData = await inspector.inspectSession(sessionId, req.correlationId);

      res.json({
        success: true,
        sessionId,
        data: sessionData,
        correlationId: req.correlationId
      });
    } catch (error) {
      res.status(500).json({
        success: false,
        error: error.message,
        correlationId: req.correlationId
      });
    }
  });
```

```
  // Endpoint to analyze database query performance
  router.get('/debug/queries', (req, res) => {
    const profiler = req.app.get('dbProfiler'); // Assume profiler is attached to app
    const analysis = profiler.getQueryAnalysis();

    res.json({
      success: true,
      queryAnalysis: analysis,
      correlationId: req.correlationId
    });
  });

  // Endpoint to simulate various error conditions for testing
  router.post('/debug/simulate-error', (req, res, next) => {
    const { errorType } = req.body;

    switch (errorType) {
      case 'validation':
        next(new ValidationError('test_field', 'invalid_value', 'must be positive'));
        break;
      case 'inventory':
        next(new InventoryError('123', 5, 2));
        break;
      case 'payment':
        next(new PaymentError('Card declined', { cardType: 'visa', lastFour: '1234' }));
        break;
      default:
        next(new Error('Generic error for testing'));
    }
  });
}
```

```
module.exports = router;
```

Milestone Checkpoints

Milestone 1 Checkpoint (Product Catalog Debugging):

- Run `npm test -- --grep "catalog"` to verify basic catalog functionality
- Test product search with `curl "http://localhost:3000/api/products?search=laptop"` and verify JSON response
- Check database queries in logs - should see structured query logging with execution times
- Inspect browser Network tab during product browsing - API calls should complete under 200ms
- Signs of problems: Missing products in results, slow page loads, error messages in console

Milestone 2 Checkpoint (Shopping Cart Debugging):

- Test cart operations: `curl -X POST http://localhost:3000/api/cart/add -d '{"productId": 1, "quantity": 2}'`
- Verify session persistence by refreshing browser and checking cart contents remain
- Check application logs for cart operation entries with correlation IDs
- Test concurrent cart updates in multiple browser tabs
- Signs of problems: Cart items disappearing, quantity updates not persisting, session errors

Milestone 3 Checkpoint (Authentication Debugging):

- Test registration: `curl -X POST http://localhost:3000/api/auth/register -d '{"email": "test@example.com", "password": "secure123"}'`
- Verify password hashing in database - passwords should be bcrypt hashes starting with "$2b$"
- Test session creation after login - should see session records in storage
- Check authentication middleware logs for protected route access
- Signs of problems: Plaintext passwords in database, session fixation, authentication bypass

Milestone 4 Checkpoint (Checkout Process Debugging):

- Test complete checkout flow with valid cart and address information
- Verify transaction atomicity - order creation and inventory update should succeed together
- Check for race condition handling with concurrent checkout attempts
- Monitor payment authorization/void flows in logs
- Signs of problems: Partial order creation, inventory overselling, payment authorization leaks

Future Extensions

> **Milestone(s):** All milestones - describes potential enhancements and how the current design accommodates future growth beyond the basic e-commerce functionality established in product catalog (Milestone 1), shopping cart (Milestone 2), user authentication (Milestone 3), and checkout process (Milestone 4)

Building an e-commerce platform is like constructing a house with a solid foundation that can support future additions. While our current system provides the essential rooms for daily living, the architectural decisions we've made create natural extension points for adding new wings, upgrading systems, and scaling to accommodate growth. The modular design patterns established across our four milestones create a flexible foundation that can evolve with business requirements.

Think of our current system as a well-planned neighborhood where each component occupies its own lot with clear boundaries, utilities run through designated channels, and zoning regulations (our interfaces) ensure new construction integrates smoothly with

existing structures. This architectural foresight enables systematic expansion without requiring complete reconstruction.

Immediate Extensions

Immediate extensions represent features that integrate seamlessly with our existing three-tier web architecture, leveraging the established data model and component interfaces without requiring fundamental architectural changes. These extensions follow the principle of least disruption while maximizing value delivery.

Product Features and Merchandising

Our current `Product` entity with its `name`, `description`, `price`, `inventory_quantity`, `category_id`, `image_url`, and `is_active` fields provides several natural extension points for enhanced merchandising capabilities. The existing product catalog component architecture can accommodate sophisticated product management features without structural modifications.

Product Reviews and Ratings can be added by extending the data model with additional entities while preserving existing relationships. The review system would introduce a `ProductReview` entity containing `id`, `product_id`, `user_id`, `rating`, `review_text`, `created_at`, and `verified_purchase` fields. This extension leverages our existing user authentication component and product catalog component, creating a natural bridge between customer experience and product data.

Product Variants and Options represent a natural evolution of our single-product model. A `ProductVariant` entity with `id`, `product_id`, `variant_type`, `variant_value`, `price_adjustment`, and `inventory_quantity` fields would enable size, color, and style variations while maintaining inventory consistency through our existing `updateInventory()` method patterns.

Wishlist Functionality parallels our shopping cart architecture, reusing the session-based state management patterns established in Milestone 2. A `Wishlist` entity with similar structure to `Cart` (`id`, `session_id`, `user_id`, `created_at`) and corresponding `WishlistItem` entities would leverage identical persistence and session management strategies.

Extension Feature	Data Model Changes	Code Changes	Integration Points
Product Reviews	Add ProductReview entity	Extend ProductService with review methods	User Authentication, Product Catalog
Product Variants	Add ProductVariant entity	Modify cart validation logic	Shopping Cart, Inventory Management
Wishlist	Add Wishlist, WishlistItem entities	Reuse cart management patterns	Session Management, User Authentication
Product Images Gallery	Extend Product.image_url to ProductImage entity	Update catalog display logic	Product Catalog, File Storage

> **Decision: Immediate Extensions Through Entity Extension**
>
> - **Context**: Current system has established patterns for data modeling, session management, and component interaction
> - **Options Considered**: Modify existing entities vs. add new entities vs. external microservices
> - **Decision**: Add new entities while preserving existing interfaces
> - **Rationale**: Maintains data consistency through existing transaction patterns while minimizing integration complexity
> - **Consequences**: Extensions integrate naturally but may increase database complexity over time

Enhanced Search and Discovery

Our current `searchProducts()` and `findWithFilters()` methods provide a foundation for sophisticated product discovery features. The existing search architecture supports immediate enhancements without requiring search infrastructure replacement.

Search Autocomplete and Suggestions can be implemented by extending the `getSuggestions()` method to include query history, popular searches, and product name matching. This enhancement reuses our existing database connections and search patterns while improving user experience through predictive text functionality.

Recently Viewed Products leverages our session management infrastructure established in the user authentication component. A `RecentlyViewed` entity with `session_id`, `product_id`, and `viewed_at` fields would track product browsing history, enabling personalized recommendations and improved product discovery.

Advanced Filtering Options extend our current category-based filtering to include price ranges, brand filtering, rating thresholds, and availability status. These filters integrate with the existing `findWithFilters()` method signature while providing more granular product discovery capabilities.

Search Enhancement	Implementation Approach	Session Integration	Performance Considerations
Autocomplete	Extend getSuggestions() with caching	Reuse existing session storage	Add search term indexing
Recently Viewed	New entity with session tracking	Leverage SessionInspector patterns	Implement automatic cleanup
Advanced Filters	Expand filter parameter object	Use existing filter validation	Add composite database indexes
Search History	Store query patterns per session	Integrate with user profiles	Implement privacy controls

Enhanced Cart Features

The shopping cart component established in Milestone 2 provides robust foundations for advanced cart management features. The existing `CartSummary` structure and cart state machine can accommodate sophisticated cart behaviors without architectural modifications.

Save for Later functionality extends our cart state machine with additional states beyond the current `empty`, `active`, `abandoned`, and `converted` states. Items can transition between active cart and saved items using the existing `CartItem` entity structure with an additional `status` field indicating `active` or `saved`.

Cart Sharing and Collaboration builds upon our session management patterns by enabling temporary cart access through shareable tokens. This feature extends the existing `transferCart()` method patterns to support multi-user cart access while maintaining security through token-based authorization.

Bulk Operations enhance the current `addItem()`, `updateQuantity()`, and `removeItem()` methods to support batch operations, improving user experience when managing multiple items simultaneously.

Architectural Extensions

Architectural extensions address scaling challenges and major feature additions that require fundamental changes to our system structure. These extensions preserve the core business logic while adapting the infrastructure to handle increased complexity and traffic.

Microservices Decomposition

Our current monolithic architecture with clearly defined component boundaries creates natural seams for microservices decomposition. The existing component interfaces serve as API contracts for service boundaries.

Service Boundaries and Communication would decompose our system into Product Catalog Service, Cart Management Service, User Authentication Service, and Order Processing Service. Each service would expose REST APIs matching our existing

component interfaces, with inter-service communication following request-response architecture patterns established in our current design.

The Product Catalog Service would encapsulate the `ProductService`, `getSuggestions()`, and `findWithFilters()` functionality within an independent deployment unit. This service would maintain its own database schema containing `Product`, `Category`, and related entities while exposing HTTP APIs matching our current catalog interface methods.

The Cart Management Service would manage shopping cart state and session persistence independently from user authentication concerns. This decomposition leverages our existing session-based state coordination patterns while enabling independent scaling of cart operations.

Database Per Service patterns would require decomposing our current unified data model into service-specific schemas. Each service maintains referential integrity within its bounded context while using eventual consistency patterns for cross-service data synchronization.

Service Boundary	Current Components	Database Entities	API Responsibilities
Product Catalog	ProductService, Search	Product, Category, ProductImage	Product listing, search, filtering
Cart Management	Cart operations, Session	Cart, CartItem, Session	Cart CRUD, inventory validation
User Management	Authentication, Sessions	User, Session, UserProfile	Registration, login, profile
Order Processing	Checkout, Payment	Order, OrderItem, Payment	Checkout flow, order creation

> **Decision: Service Decomposition Along Component Boundaries**
>
> - **Context**: Current monolithic architecture has well-defined component interfaces that map naturally to service boundaries
> - **Options Considered**: Decompose by data entities vs. decompose by business capabilities vs. maintain monolithic architecture
> - **Decision**: Decompose along existing component boundaries with API contracts matching current interfaces
> - **Rationale**: Existing component interfaces provide proven API contracts while component isolation enables independent deployment
> - **Consequences**: Services can scale independently but introduces network communication overhead and distributed systems complexity

Caching and Performance Optimization

Performance scaling requires introducing caching layers and optimization strategies while preserving data consistency guarantees established in our current design. The existing query delegation model and session-based state coordination provide integration points for caching mechanisms.

Multi-Level Caching Strategy would introduce application-level caching for product catalog data, session-level caching for user-specific information, and database-level caching for frequently accessed queries. This caching hierarchy respects our existing data consistency requirements while improving response times.

Product catalog caching would cache `ProductListResponse` and `ProductDetailResponse` objects at the application layer, with cache invalidation triggered by inventory updates through the existing `updateInventory()` method. This approach maintains inventory consistency while accelerating product browsing operations.

Cart caching would leverage our existing session management infrastructure to cache `CartSummary` objects, reducing database queries for frequent cart operations while ensuring cart persistence across browser sessions remains intact.

Database Optimization would introduce read replicas for catalog browsing operations while maintaining write operations against the primary database. This approach leverages our existing query delegation model by routing read operations to replicas while

preserving transactional operations against the primary database.

Caching Layer	Data Types Cached	Invalidation Strategy	Consistency Requirements
Application	ProductListResponse, CategoryHierarchy	Product updates, inventory changes	Eventually consistent
Session	CartSummary, UserProfile	Session expiration, user actions	Session-consistent
Database	Query results, aggregations	Write operations, scheduled refresh	Read-after-write consistent
Content Delivery	Product images, static assets	File updates, manual invalidation	Eventually consistent

Advanced Order Management

Order processing extensions build upon the transactional operations and ACID transaction principles established in our checkout process component. These extensions maintain atomicity while adding sophisticated order lifecycle management.

Order Status Tracking extends our current `Order` entity with additional status fields and audit trails. An `OrderStatusHistory` entity with `id`, `order_id`, `status`, `timestamp`, `notes`, and `updated_by` fields would track order progression through fulfillment workflows while maintaining the existing order creation atomicity.

Inventory Reservation System would enhance our current pessimistic locking patterns with time-bounded reservations. This system would introduce `InventoryReservation` entities that automatically expire, preventing abandoned checkouts from indefinitely blocking inventory while maintaining the inventory consistency guarantees established in our checkout process.

Advanced Payment Processing would extend our current payment integration stub with multiple payment methods, payment installments, and refund processing. This extension maintains the existing `authorizePayment()` and `capturePayment()` interface contracts while supporting diverse payment workflows.

Order Extension	Current Foundation	New Entities	Integration Complexity
Status Tracking	Order entity, transactional operations	OrderStatusHistory	Low - extends existing patterns
Inventory Reservations	Pessimistic locking, checkout sessions	InventoryReservation	Medium - requires timeout management
Advanced Payments	Payment stub, authorization patterns	Payment, PaymentMethod	Medium - external service integration
Return Processing	Order entity, inventory updates	Return, ReturnItem	High - reverse inventory transactions

Design Accommodations

Our current architecture incorporates several design patterns and architectural decisions that specifically accommodate future modifications without requiring complete system reconstruction. These accommodations represent conscious trade-offs made during initial design to preserve extensibility.

Interface-Based Component Isolation

The component isolation established across our four milestones creates natural extension points through well-defined interfaces. Each component exposes its functionality through specific method signatures that serve as stable contracts for future enhancements.

The `ProductService` interface with methods like `findWithFilters()`, `searchProducts()`, and `updateInventory()` provides extension hooks for enhanced search algorithms, recommendation engines, and inventory management systems. New implementations can replace existing logic while maintaining compatibility with cart validation and checkout processes.

Session management through the `Session` entity and methods like `validateSession()` and `createSession()` enables authentication extensions including OAuth integration, multi-factor authentication, and single sign-on capabilities without modifying cart or checkout components that depend on session validation.

Dependency Injection Patterns established in our component design enable runtime configuration of service implementations. The `OrderService` with its `db`, `paymentService`, `inventoryService`, and `logger` dependencies can accommodate alternative implementations for payment processing, inventory management, and logging without requiring changes to order processing logic.

Interface Contract	Current Implementation	Extension Capabilities	Compatibility Requirements
ProductService methods	Database queries	Search engines, external catalogs	Maintain response object structure
Cart operations	Session storage	Redis, database clustering	Preserve state machine transitions
Authentication methods	bcrypt, session cookies	OAuth, JWT, multi-factor	Maintain session validation interface
Payment interface	Authorization stub	Multiple processors, wallets	Preserve transaction semantics

Database Schema Flexibility

Our data model accommodations support schema evolution through several strategic design decisions made during the initial modeling phase. These decisions prioritize extensibility while maintaining referential integrity and data constraints.

Extensible Attribute Storage can be accommodated through additional columns on existing entities or through attribute tables that extend core entities without modifying existing relationships. The `Product` entity structure supports additional attributes through supplementary tables that maintain foreign key relationships to the core product data.

Category Hierarchy Extensions are supported through the existing `Category` entity design with `parent_id` and `path` fields. This hierarchical structure accommodates arbitrary nesting depths and category reorganization without breaking existing product associations or filtering logic.

Audit Trail Capabilities can be added through supplementary audit tables that track changes to core entities without modifying existing table structures. These audit tables would capture entity snapshots and change metadata while preserving the performance characteristics of operational tables.

> **Key Design Principle: Schema Evolution Strategy** The database schema design prioritizes additive changes over destructive modifications. New features add tables and columns rather than modifying existing structures, preserving compatibility with existing application code while enabling gradual feature rollout.

Schema Extension Pattern	Implementation Strategy	Backward Compatibility	Migration Complexity
Additional Columns	ALTER TABLE statements	Full compatibility	Low
Supplementary Tables	Foreign key relationships	No impact on existing queries	Low
Attribute Extensions	Key-value or JSON columns	Query modifications required	Medium
Schema Versioning	Migration scripts with rollback	Version-specific compatibility	High

API Versioning and Evolution

The REST API patterns established across our milestone implementations create a foundation for API versioning that accommodates breaking changes while maintaining client compatibility. The existing response envelope pattern with `ProductListResponse`, `CartSummary`, and `OrderConfirmation` structures provides versioning accommodation through response structure evolution.

Version Header Strategy would introduce API versioning through HTTP headers while maintaining existing endpoint URLs. Clients would specify their expected API version through `Accept-Version` headers, enabling the same endpoint to serve multiple response formats based on client capabilities.

Response Envelope Evolution leverages our existing envelope pattern where responses include metadata objects alongside core data. New API versions can extend metadata sections while preserving core data structure compatibility, enabling gradual client migration to enhanced features.

Backward Compatibility Windows would maintain support for previous API versions for defined periods, allowing client applications to migrate gradually while new features become available immediately to updated clients.

Component Extensibility Patterns

The component architecture established across our milestones incorporates several patterns that facilitate extension without requiring core logic modifications. These patterns follow the principle of composition over inheritance while maintaining clear separation of concerns.

Middleware Extension Points in our Express application setup enable cross-cutting concerns like analytics tracking, audit logging, and security enhancements to be added without modifying existing route handlers. The `setupMiddleware()` method can accommodate additional middleware while preserving the existing request-response flow.

Event Hooks and Notifications can be added through observer patterns that trigger on existing operations like cart modifications, order creation, and inventory updates. These hooks would enable email notifications, analytics tracking, and external system integration while maintaining the transactional integrity of core operations.

Plugin Architecture Foundations exist in our component isolation patterns where services receive dependencies through constructor injection. This pattern enables plugin-style extensions where core services can be enhanced with additional capabilities through decorator patterns or composition.

> **Architectural Insight: Extension Through Composition** Rather than modifying existing components, extensions should compose new capabilities around existing interfaces. This approach preserves the stability of tested code while enabling rapid feature development through proven building blocks.

Extension Pattern	Current Foundation	Implementation Strategy	Compatibility Impact
Middleware Hooks	Express middleware stack	Insert additional middleware	Zero impact on existing routes
Event Observers	Component method calls	Add notification calls to existing methods	Minimal impact on core logic
Service Decorators	Dependency injection	Wrap existing services with enhanced versions	Transparent to dependent components
Configuration Extensions	Environment variables	Add configuration sections	Backward compatible defaults

Performance Scaling Accommodations

The current architecture decisions anticipate performance scaling challenges through several design accommodations that can be activated as traffic demands increase. These accommodations leverage existing patterns while introducing performance optimizations.

Database Connection Pooling can be integrated into our existing `Database` component without modifying the `getConnection()` interface contract. Connection pooling would improve concurrent request handling while maintaining the

existing database access patterns established across all components.

Horizontal Scaling Preparations are accommodated through our session-based authentication and cart management patterns. Since session data is encapsulated within defined entities and accessed through specific methods, session storage can be migrated to shared storage systems like Redis without modifying application logic.

Caching Integration Points exist throughout our component interfaces where expensive operations like `findWithFilters()`, `searchProducts()`, and `getCartContents()` can be enhanced with caching layers while preserving existing method signatures and response formats.

Advanced Feature Integration

Advanced features require more substantial architectural considerations while building upon the foundations established in our current design. These features represent significant functional expansions that leverage existing patterns while introducing new complexity dimensions.

Multi-Tenant and Marketplace Features

Marketplace functionality transforms our single-store architecture into a platform supporting multiple vendors while preserving the user experience patterns established in our current implementation. This transformation requires systematic extension of our data model and component interfaces.

Vendor Management would introduce a `Vendor` entity with `id`, `name`, `contact_email`, `status`, and `commission_rate` fields that extends our existing `Product` entity through a `vendor_id` foreign key relationship. This extension preserves existing product catalog functionality while enabling vendor-specific product management and commission tracking.

Multi-Vendor Cart Management builds upon our existing cart state machine by adding vendor grouping logic within cart operations. The existing `CartItem` entity structure supports vendor separation through product relationships, enabling vendor-specific subtotals and shipping calculations while maintaining unified cart management interfaces.

Vendor Dashboard Integration would extend our user authentication component with role-based access control, adding vendor-specific authentication flows and management interfaces while preserving customer authentication patterns.

Marketplace Feature	Data Model Extensions	Component Modifications	Integration Challenges
Vendor Management	Vendor entity, Product.vendor_id	Extend ProductService with vendor filtering	Access control complexity
Split Payments	VendorPayment entity	Modify payment processing for splits	Payment processor integration
Vendor Analytics	VendorSales aggregation entity	Add reporting interfaces	Data privacy requirements
Review Moderation	ReviewModeration entity	Extend review workflows	Content management complexity

Advanced Inventory Management

Sophisticated inventory management builds upon our current `updateInventory()` method and pessimistic locking patterns while introducing warehouse management and supply chain integration capabilities.

Multi-Location Inventory would extend our `Product` entity through an `InventoryLocation` entity containing `product_id`, `location_id`, `quantity`, and `reserved_quantity` fields. This extension maintains our existing inventory validation patterns while enabling location-specific availability and shipping optimization.

Automated Reordering can be integrated into our existing inventory update workflows by adding threshold monitoring to the `updateInventory()` method. When inventory levels drop below defined thresholds, automated purchase orders could be

generated while maintaining real-time inventory accuracy.

Inventory Forecasting would leverage our `OrderItem` historical data to predict demand patterns, integrating with inventory updates to optimize stock levels while preserving the immediate inventory validation required for our checkout process.

Payment and Financial Extensions

Advanced payment capabilities build upon our current payment authorization patterns while introducing sophisticated financial management features that maintain the transactional integrity established in our checkout process.

Multiple Payment Methods extend our existing `authorizePayment()` interface to support credit cards, digital wallets, buy-now-pay-later options, and cryptocurrency payments. Each payment method would implement the same authorization and capture interface contracts while handling method-specific processing requirements.

Subscription and Recurring Payments would introduce `Subscription` and `RecurringPayment` entities that leverage our existing order creation patterns for automated billing cycles. These entities would integrate with our user authentication and payment authorization systems while adding temporal scheduling capabilities.

Financial Reporting and Analytics would aggregate data from our existing `Order`, `OrderItem`, and payment entities to provide business intelligence capabilities while maintaining the privacy and security constraints established in our user authentication component.

Payment Extension	Technical Foundation	Integration Points	Compliance Requirements
Digital Wallets	Payment authorization interface	User authentication, checkout	PCI DSS compliance
Subscription Billing	Order creation patterns	User management, inventory	Recurring payment regulations
Refund Processing	Payment capture reversal	Order management, inventory	Financial audit requirements
Split Payments	Order total calculation	Multi-vendor, commission	Tax calculation complexity

International and Localization Extensions

Internationalization capabilities build upon our existing data model and user management systems while introducing currency, language, and regional customization features.

Multi-Currency Support would extend our `Product` entity pricing through a `ProductPrice` entity containing `product_id`, `currency_code`, `price`, and `effective_date` fields. This extension maintains our price precision practices using cents while supporting multiple currencies through currency-specific price storage.

Localized Content can be accommodated through supplementary content entities that provide translated product descriptions, category names, and interface text while maintaining the existing product catalog and category hierarchy structures.

Regional Compliance would extend our checkout process component with region-specific tax calculation, shipping restrictions, and regulatory compliance while preserving the existing order processing algorithm and address validation patterns.

Implementation Strategy and Migration Paths

The transition from our current basic e-commerce system to advanced capabilities requires careful planning that preserves operational stability while introducing new features incrementally. Our architectural accommodations enable systematic enhancement through defined migration paths.

Feature Flag Patterns

Feature flags provide a mechanism for gradual feature rollout while maintaining system stability. Our current component structure supports feature flag integration through configuration-driven behavior modification without requiring code changes for each feature toggle.

Component-Level Feature Flags would control major feature availability through environment variables that modify component behavior at runtime. For example, advanced search features could be enabled through configuration flags that activate enhanced search algorithms while maintaining backward compatibility with basic search functionality.

User-Level Feature Flags would leverage our existing user authentication infrastructure to provide personalized feature access, enabling beta testing of new capabilities with selected user groups while preserving stable functionality for general users.

Database Migration Strategy

Schema evolution requires careful planning to maintain data integrity while accommodating new features. Our current database schema design supports additive migrations that preserve existing functionality while introducing new capabilities.

Additive Schema Changes follow the principle of adding new tables and columns rather than modifying existing structures. New features introduce supplementary entities that extend core functionality through foreign key relationships while preserving existing query patterns.

Zero-Downtime Migrations can be achieved through our existing transaction atomicity patterns by applying schema changes during maintenance windows and using backward-compatible migration scripts that support both old and new schema versions during transition periods.

Data Migration Patterns would leverage our existing entity relationships to migrate data from simple structures to complex structures while maintaining referential integrity throughout the migration process.

Long-Term Architectural Evolution

The long-term evolution path for our e-commerce platform anticipates growth from a learning project to a production-capable system while preserving the educational value of the current implementation.

Scalability Transformation

Horizontal Scaling Readiness exists in our session-based authentication and stateless API design patterns. The current architecture can accommodate load balancer distribution and multiple application instances without requiring fundamental changes to request handling or session management.

Database Sharding Preparation can build upon our existing entity relationships and data access patterns. Product catalog data can be sharded by category or geographic region while preserving the existing search and filtering interfaces through query routing layers.

Event-Driven Architecture Migration would transform our current request-response architecture into an event-driven system while maintaining existing API contracts for client applications. Internal component communication would evolve to use asynchronous messaging while preserving synchronous behavior for client-facing operations.

Production-Grade Reliability

Monitoring and Observability would extend our existing structured logging patterns with comprehensive metrics collection, distributed tracing, and alerting capabilities while maintaining the debugging capabilities established in our current design.

High Availability Patterns would introduce redundancy and failover capabilities while preserving our existing graceful degradation and circuit breaker pattern implementations. These patterns ensure system availability during partial failures while maintaining data consistency guarantees.

Security Hardening would enhance our existing password hashing and session management security with additional protections like rate limiting, intrusion detection, and advanced threat monitoring while preserving the security foundations established in our authentication component.

Production Capability	Current Foundation	Enhancement Strategy	Operational Impact
Load Balancing	Stateless API design	Add reverse proxy layer	Improved request distribution
Database Clustering	Transaction patterns	Add read replicas	Enhanced read performance
Security Monitoring	Session validation	Add security event logging	Improved threat detection
Automated Scaling	Component isolation	Add container orchestration	Dynamic resource allocation

> **Long-Term Vision: Educational Foundation to Production System** The current architecture serves dual purposes as a comprehensive learning platform and a foundation for production system evolution. The design accommodations ensure that educational projects can transition to commercial viability while preserving the learning patterns that make the system accessible to developers at all skill levels.

Implementation Guidance

The extension implementation strategy leverages modern JavaScript/Node.js patterns while building upon the Express.js and SQLite foundations established in our current system. This guidance provides concrete approaches for implementing the most valuable immediate extensions.

Technology Recommendations for Extensions

Extension Category	Simple Option	Advanced Option	Integration Complexity
Caching	Node.js Map objects	Redis with node-redis client	Low to Medium
Search Enhancement	SQLite FTS	Elasticsearch with @elastic/elasticsearch	Medium to High
File Storage	Local filesystem	AWS S3 with aws-sdk	Low to Medium
Email Notifications	nodemailer with SMTP	SendGrid API integration	Low
Analytics	Custom logging	Google Analytics 4 integration	Medium
Payment Processing	Stripe Elements	Multiple processors with adapter pattern	Medium to High

Recommended Extension File Structure

Building upon our existing project organization, extensions should follow consistent module patterns that preserve the separation of concerns established in our current architecture:

```
project-root/
  src/
    extensions/
      reviews/
        ReviewService.js          ← business logic
        ReviewAPI.js              ← HTTP endpoints
        review-routes.js          ← Express route definitions
        review-validation.js      ← input validation
      wishlist/
        WishlistService.js        ← cart-like functionality
        wishlist-routes.js        ← API endpoints
      notifications/
        EmailService.js           ← email delivery
        NotificationHooks.js      ← event observers
      caching/
        CacheManager.js           ← cache coordination
        cache-middleware.js       ← Express caching
    core/                         ← existing components
      products/
      cart/
      auth/
      checkout/
```

Immediate Extension Starter Code

Product Reviews Extension Infrastructure:

```javascript
// src/extensions/reviews/ReviewService.js

class ReviewService {

    constructor(database, userService) {

        this.db = database;

        this.userService = userService;

    }


    async createReview(productId, userId, rating, reviewText) {

        // TODO 1: Validate user has purchased this product using OrderItem queries

        // TODO 2: Check for existing review from this user for this product

        // TODO 3: Validate rating is between 1 and 5

        // TODO 4: Create review record with verified_purchase flag

        // TODO 5: Update product aggregate rating using transaction

        // Hint: Use transaction to ensure rating consistency

    }


    async getProductReviews(productId, pagination) {

        // TODO 1: Query reviews for product with user information joins

        // TODO 2: Apply pagination using LIMIT and OFFSET

        // TODO 3: Calculate review summary statistics

        // TODO 4: Return paginated response with review metadata

        // Hint: Join with User table but protect email privacy

    }


    async updateReview(reviewId, userId, updates) {

        // TODO 1: Verify review ownership by checking userId match

        // TODO 2: Validate updated rating and text content

        // TODO 3: Update review record with modification timestamp

        // TODO 4: Recalculate product aggregate rating

        // Hint: Use pessimistic locking during rating recalculation

    }

}
```

JAVASCRIPT

Wishlist Extension with Cart Pattern Reuse:

```javascript
// src/extensions/wishlist/WishlistService.js

class WishlistService {

    constructor(database, sessionService) {

        this.db = database;

        this.sessionService = sessionService;

    }


    async addToWishlist(sessionId, productId) {

        // TODO 1: Get or create wishlist for session using cart patterns

        // TODO 2: Verify product exists and is active

        // TODO 3: Check if product already in wishlist

        // TODO 4: Insert wishlist item record

        // TODO 5: Return updated wishlist summary

        // Hint: Reuse session validation patterns from cart component

    }


    async moveToCart(sessionId, wishlistItemId, quantity) {

        // TODO 1: Validate wishlist item ownership

        // TODO 2: Check product inventory availability

        // TODO 3: Add item to cart using existing addItem method

        // TODO 4: Remove item from wishlist

        // TODO 5: Return cart and wishlist summaries

        // Hint: Use transaction to ensure atomicity between wishlist and cart

    }

}
```

JAVASCRIPT

Caching Layer Implementation

Application-Level Caching Manager:

```javascript
// src/extensions/caching/CacheManager.js

class CacheManager {

    constructor(cacheStore = new Map()) {

        this.cache = cacheStore;

        this.defaultTTL = 300000; // 5 minutes in milliseconds

    }


    async get(key, fetchFunction, ttlMs = this.defaultTTL) {

        // TODO 1: Check if key exists in cache and is not expired

        // TODO 2: If cache hit, return cached value

        // TODO 3: If cache miss, execute fetchFunction to get fresh data

        // TODO 4: Store result in cache with expiration timestamp

        // TODO 5: Return fresh data to caller

        // Hint: Use fetchFunction.apply() for dynamic data loading

    }


    invalidatePattern(pattern) {

        // TODO 1: Iterate through cache keys matching pattern

        // TODO 2: Remove matching entries from cache

        // TODO 3: Log invalidation for debugging

        // Hint: Use RegExp for pattern matching cache keys

    }


    async wrapProductService(productService) {

        // TODO 1: Create proxy object with same interface as ProductService

        // TODO 2: Implement cache-aware versions of findById, findWithFilters

        // TODO 3: Add cache invalidation to updateInventory calls

        // TODO 4: Return enhanced service with transparent caching

        // Hint: Use Proxy object for method interception

    }

}
```

JAVASCRIPT

Extension Integration Patterns

Event Hook System for Cross-Component Communication:

JAVASCRIPT

```javascript
// src/extensions/notifications/NotificationHooks.js

class NotificationHooks {

    constructor(emailService, analyticsService) {

        this.emailService = emailService;

        this.analytics = analyticsService;

    }


    async onOrderCreated(orderConfirmation, userEmail) {

        // TODO 1: Send order confirmation email asynchronously

        // TODO 2: Track order completion event in analytics

        // TODO 3: Update user purchase history

        // TODO 4: Handle notification failures gracefully

        // TODO 5: Log notification attempts for debugging

        // Hint: Use Promise.allSettled for parallel notification delivery

    }


    async onInventoryLow(productId, currentQuantity, threshold) {

        // TODO 1: Query vendor contact information for product

        // TODO 2: Generate low inventory alert email

        // TODO 3: Log inventory alert for supply chain management

        // TODO 4: Update inventory monitoring dashboard

        // Hint: Don't block inventory updates for notification failures

    }


    integrateWithOrderService(orderService) {

        // TODO 1: Wrap orderService.confirmOrder with notification hooks

        // TODO 2: Ensure notifications don't affect order processing success

        // TODO 3: Add retry logic for failed notifications

        // TODO 4: Provide fallback communication methods

        // Hint: Decorator pattern preserves original interface

    }

}
```

Milestone Checkpoints for Extension Development

Extension Development Verification Process:

1. **Interface Compatibility Testing**: Verify that extensions maintain existing API contracts by running existing test suites against extended components
2. **Performance Impact Assessment**: Measure response time differences between base functionality and extended functionality to ensure acceptable performance overhead
3. **Data Consistency Validation**: Confirm that extensions preserve transactional integrity and data constraints established in the base system
4. **Session Management Integration**: Test extension functionality with both authenticated and anonymous sessions to ensure session handling consistency
5. **Error Handling Verification**: Validate that extensions follow established error handling patterns and don't introduce unhandled exception scenarios

Extension Testing Strategy:

- **Backward Compatibility Tests**: Ensure existing functionality works unchanged after extension installation
- **Integration Boundary Tests**: Verify data flows correctly between core components and extensions
- **Performance Regression Tests**: Confirm extensions don't degrade core system performance beyond acceptable thresholds
- **Security Impact Tests**: Validate extensions don't introduce security vulnerabilities or compromise existing authentication and authorization

Migration Path Implementation

Gradual Feature Rollout Strategy:

```javascript
// src/extensions/feature-flags/FeatureManager.js

class FeatureManager {

    constructor(configService, userService) {

        this.config = configService;

        this.userService = userService;

    }


    async isFeatureEnabled(featureName, sessionId, defaultValue = false) {

        // TODO 1: Check global feature flag configuration

        // TODO 2: Check user-specific feature flag overrides

        // TODO 3: Implement percentage-based rollout logic

        // TODO 4: Log feature flag evaluations for monitoring

        // TODO 5: Return boolean indicating feature availability

        // Hint: Use consistent hashing for percentage rollouts

    }


    async enableFeatureForUser(featureName, userId, enabled) {

        // TODO 1: Validate feature name exists in configuration

        // TODO 2: Store user-specific feature override

        // TODO 3: Log feature flag change for audit

        // TODO 4: Clear any cached feature evaluations for user

        // Hint: Use user preferences table for flag storage

    }

}
```

The extension implementation guidance provides a roadmap for systematic enhancement while preserving the educational value and operational stability of the core e-commerce platform. Each extension builds upon established patterns and interfaces, ensuring consistent development practices and architectural coherence as the system evolves from a learning project to a production-capable platform.

Glossary

> **Milestone(s):** All milestones - provides definitions for technical terms, domain concepts, and acronyms that support understanding across product catalog (Milestone 1), shopping cart (Milestone 2), user authentication (Milestone 3), and checkout process (Milestone 4)

The e-commerce domain introduces numerous technical concepts, architectural patterns, and specialized terminology that span web development, database design, security, and business logic. Understanding these terms precisely is essential for implementing a robust e-commerce system and communicating effectively about its design and operation.

This glossary serves as both a reference for terminology used throughout the design document and a learning resource for developers new to e-commerce systems. Each definition includes context about how the concept applies to our specific e-commerce architecture and implementation.

Technical Architecture Terms

Three-tier web architecture refers to the structural organization of web applications into three distinct layers: the presentation layer (frontend user interface), the application layer (business logic and API), and the data layer (database and persistence). Each tier has specific responsibilities and communicates only with adjacent tiers, creating clear separation of concerns and enabling independent scaling and maintenance.

Monolithic architecture describes a software design approach where all functionality is contained within a single deployable application. In our e-commerce context, this means the product catalog, shopping cart, user authentication, and checkout processes all exist within the same codebase and share the same database connection, simplifying development and deployment for learning purposes.

Microservices architecture represents an alternative approach where functionality is decomposed into independent services that communicate over network protocols. While our design uses a monolithic approach for simplicity, understanding microservices helps developers appreciate the trade-offs and prepare for future scaling considerations.

Component responsibilities define the specific duties and data ownership of each major system component. Clear responsibility boundaries prevent overlap, reduce coupling, and make the system easier to understand and maintain. In our architecture, each component owns specific database entities and provides well-defined interfaces for other components to interact with its data.

Layered interaction model describes how system components communicate through defined interfaces rather than direct access. The presentation layer calls application layer methods, which in turn interact with the data layer, but presentation layer code never directly queries the database.

Dependency injection patterns refer to techniques where components receive their dependencies from external sources rather than creating them internally. This approach improves testability by allowing mock dependencies during testing and reduces coupling between components.

Request-response architecture describes the communication pattern where clients initiate operations by sending HTTP requests to server endpoints, which process the request and return structured responses. This stateless approach simplifies server design and enables horizontal scaling.

Data Management Terms

Entity relationships describe the connections between database tables through foreign key constraints. These relationships enforce referential integrity and enable complex queries that span multiple entities. Our e-commerce system uses relationships like User to Order (one-to-many) and Order to OrderItem (one-to-many).

Referential integrity ensures that foreign key relationships remain valid across all database operations. When a User is deleted, the system must handle their associated Orders appropriately, either by preventing deletion or cascading the deletion to related records.

Data constraints are database rules that enforce business logic and data validity at the storage layer. These include check constraints (ensuring positive prices), unique constraints (preventing duplicate emails), and not-null constraints (requiring essential fields).

Hierarchical structure refers to tree-like data organization where entities have parent-child relationships. Our Category entity uses this structure with a parent_id field, enabling nested product categories like Electronics > Computers > Laptops.

Transaction atomicity guarantees that multi-step database operations either complete entirely or fail completely, preventing partial updates that could leave data in an inconsistent state. Checkout processing relies heavily on atomicity to ensure inventory, orders, and payments remain synchronized.

Pessimistic locking prevents concurrent access to database records during transactions by acquiring exclusive locks. The `SELECT FOR UPDATE` SQL construct implements this pattern, ensuring that inventory checks and updates happen atomically without interference from other concurrent operations.

Optimistic locking detects conflicts through version comparison rather than preventing concurrent access. This approach assumes conflicts are rare and handles them when they occur, typically through version fields that increment with each update.

ACID transaction principles define four properties that database transactions must satisfy: Atomicity (all operations succeed or fail together), Consistency (data remains valid), Isolation (concurrent transactions don't interfere), and Durability (committed changes survive system failures).

Session and State Management Terms

Session persistence maintains user state across HTTP requests and browser sessions. Since HTTP is stateless, web applications must store user context (authentication status, cart contents, preferences) in a way that survives page refreshes and temporary network interruptions.

Session-based authentication stores user authentication status on the server using session identifiers sent to clients as HTTP cookies. This approach contrasts with token-based authentication and provides better security for web applications by keeping sensitive session data server-side.

Session fixation is a security vulnerability where attackers predetermine session identifiers to hijack user sessions. Proper session management prevents this by generating new session IDs after authentication and using secure random generation algorithms.

Sliding expiration extends session timeouts based on user activity, providing a balance between security (inactive sessions expire) and user experience (active users don't need to repeatedly log in).

Cart state machine defines the valid states a shopping cart can exist in (empty, active, abandoned, converted) and the allowed transitions between states. This formal model helps prevent invalid operations and ensures consistent cart behavior.

Session expiration automatically removes inactive sessions to free server resources and improve security. Our checkout sessions include explicit timeouts to prevent indefinite resource usage and encourage timely purchase decisions.

Stateful session management maintains user context on the server between HTTP requests, enabling complex multi-step processes like checkout flows that span multiple page interactions.

Security and Authentication Terms

Password hashing transforms plaintext passwords into irreversible cryptographic digests using algorithms like bcrypt. This ensures that even if the database is compromised, original passwords cannot be recovered, only brute-force attacked.

Bcrypt is an adaptive password hashing function that includes built-in salt generation and configurable computational cost. The cost factor can be increased over time to maintain security as computing power increases.

Timing attack exploits differences in response times to extract sensitive information. Authentication systems must use constant-time comparison functions to prevent attackers from determining valid usernames based on response timing differences.

Salt rounds determine the computational cost of bcrypt hashing. Higher salt rounds increase security by making brute-force attacks more expensive but also increase server CPU usage during authentication operations.

Authentication middleware are Express.js functions that execute before route handlers to verify user authentication status. This pattern centralizes authentication logic and ensures consistent security enforcement across protected endpoints.

Session secret is a cryptographic key used to sign session cookies, preventing clients from tampering with session data. This secret must be kept confidential and rotated periodically for optimal security.

Security layering implements multiple validation steps and cryptographic protections to provide defense in depth. Our authentication system combines password hashing, session signing, HTTPS transport, and input validation.

E-commerce Business Logic Terms

Inventory consistency maintains accurate stock levels across concurrent operations. When multiple users attempt to purchase the same product simultaneously, the system must prevent overselling while providing good user experience.

Price precision ensures accurate monetary calculations by avoiding floating-point arithmetic errors. Storing prices in cents as integers eliminates rounding errors that could accumulate over many transactions.

Price staleness occurs when cart items reflect outdated pricing information between addition and checkout. Systems must decide whether to honor original prices or update to current prices during the checkout process.

Inventory reservation temporarily holds stock for users during checkout processes, preventing other customers from purchasing reserved items while ensuring inventory is released if checkout is abandoned.

Order processing algorithm defines the step-by-step procedure for converting shopping cart contents into confirmed orders, including inventory validation, payment processing, and order record creation.

Payment authorization reserves funds on a payment method without immediately charging the customer. This two-phase approach allows inventory validation before money changes hands and enables easy cancellation if problems arise.

Checkout session isolation separates the checkout process from the active shopping cart to prevent interference during multi-step checkout flows. Users can continue browsing and adding items while completing purchases.

Address validation verifies shipping address format and deliverability to prevent order fulfillment problems. This can range from simple format checking to integration with postal service databases.

Performance and Scalability Terms

N+1 queries represent a common performance antipattern where loading a list of records triggers additional queries for each record's related data. Loading 10 products with their categories might execute 1 + 10 = 11 queries instead of a single optimized join query.

Query delegation model centralizes database interaction logic in dedicated service layers while keeping HTTP route handlers focused on request/response formatting. This separation enables query optimization and caching without affecting API interfaces.

Circuit breaker pattern prevents cascading failures by automatically failing fast when downstream services become unavailable. After detecting repeated failures, the circuit breaker stops sending requests for a timeout period before attempting recovery.

Exponential backoff is a retry strategy that increases delays between attempts to prevent overwhelming already-stressed systems. Failed operations wait progressively longer (1s, 2s, 4s, 8s) before retrying.

Graceful degradation maintains core functionality during partial system failures. If product images become unavailable, the catalog should still display product information with placeholder images rather than failing completely.

Horizontal scaling increases system capacity by adding more servers rather than upgrading existing hardware. Stateless application design enables horizontal scaling by allowing load distribution across multiple server instances.

Eventual consistency accepts that data might be temporarily inconsistent across system components but will converge to consistency over time. This pattern enables higher performance and availability in distributed systems.

Error Handling and Recovery Terms

Operational errors are expected runtime conditions that applications must handle gracefully, such as invalid user input, network timeouts, or insufficient inventory. These contrast with programming errors (bugs) that require code fixes.

Compensating transactions are operations that undo partially completed business processes when later steps fail. If payment authorization succeeds but inventory allocation fails, the compensation transaction voids the payment authorization.

Race conditions are timing-dependent bugs where concurrent operations produce inconsistent results. Two users purchasing the last item simultaneously might both see availability, leading to overselling without proper concurrency control.

Atomic state transitions ensure that complex state changes either complete entirely or fail entirely. Order creation must update inventory, create order records, and process payments atomically to prevent partial completion.

Error correlation links error symptoms to root causes through structured logging and correlation identifiers. When checkout fails, logs should connect the user's error message to specific database errors or payment gateway responses.

Testing and Quality Assurance Terms

Test pyramid describes a testing strategy with many fast unit tests at the base, fewer integration tests in the middle, and few expensive end-to-end tests at the top. This balance provides good coverage while maintaining reasonable test execution times.

Unit testing validates individual functions in isolation using mocked dependencies. Testing the `hashPassword` function should focus on hash generation without requiring database connections or external services.

Integration testing validates component interactions with real dependencies like database connections. Testing cart operations should use actual database tables to verify that add/remove operations work correctly with persistence.

End-to-end testing validates complete user workflows through browser automation. Testing checkout should simulate actual user interactions from product selection through order confirmation.

Test fixtures provide consistent sample data across test cases. Using the same set of test products, users, and orders ensures reproducible test results and simplifies test maintenance.

Mocking replaces external dependencies with controlled test doubles that provide predictable responses. Payment gateway integration tests should mock actual payment processing to avoid charging real credit cards.

Test isolation ensures that test cases don't interfere with each other by cleaning up data and resetting system state. Each test should start with a known clean state and leave the system ready for subsequent tests.

Happy path testing validates normal successful workflows where everything works as expected. These tests verify that core functionality operates correctly under ideal conditions.

Edge case testing validates boundary conditions and unusual scenarios that might reveal bugs. Testing cart operations with zero quantities, negative prices, or extremely long product names helps ensure robust error handling.

Concurrency testing validates correct behavior when multiple operations execute simultaneously. Testing inventory management with concurrent purchase attempts ensures the system prevents overselling.

Debugging and Development Terms

Structured logging uses consistent formats and includes correlation identifiers to connect related log entries across system components. When debugging checkout failures, structured logs enable tracing the complete request flow.

Correlation identifier is a unique ID that links related log entries across multiple system layers. A single user request might generate log entries in authentication, cart management, and order processing components.

Request tracing follows individual requests through multiple system layers to diagnose performance bottlenecks or error sources. Slow checkout processes can be traced to identify whether delays occur in database queries, payment processing, or other

components.

Performance profiling measures execution time and resource usage to identify optimization opportunities. Database query profiling might reveal that product search queries need indexing improvements.

Session debugging involves inspecting session state and lifecycle to diagnose authentication or cart persistence issues. Session inspection tools help developers understand why users might be unexpectedly logged out or losing cart contents.

Query profiling analyzes database query performance to identify slow queries, missing indexes, or inefficient joins. Product catalog performance problems often stem from poorly optimized search queries.

Contextual logging includes relevant business context in log entries, such as user IDs, product IDs, and order numbers. This additional context helps developers quickly understand the business impact of technical issues.

Extension and Future Growth Terms

Component isolation ensures that system components have well-defined interfaces and minimal coupling, enabling independent modification and extension. Adding product reviews should not require changes to checkout processing logic.

Interface-based extension adds new functionality through stable contracts rather than modifying existing code. New payment methods can be added by implementing the payment interface without changing order processing logic.

Additive schema changes evolve database structure through new tables and columns rather than modifying existing structures. This approach minimizes migration risks and maintains backward compatibility.

Event-driven architecture enables asynchronous communication between components through event publishing. Order completion events can trigger inventory updates, email notifications, and analytics tracking without coupling these concerns to checkout logic.

Feature flag patterns enable runtime feature toggling through configuration, allowing gradual rollout of new functionality and easy rollback if problems arise. New checkout flows can be tested with small user percentages before full deployment.

Database and SQL Terms

Primary key is an auto-incrementing unique identifier that provides fast record access and enables foreign key relationships. Every entity in our system uses integer primary keys for simplicity and performance.

Foreign key establishes referential integrity constraints between related tables. The `user_id` field in the Order table references the `id` field in the User table, ensuring orders can't exist without valid users.

Check constraint enforces business rules at the database level, such as ensuring product prices are positive or inventory quantities are non-negative. These constraints provide data quality guarantees even if application logic has bugs.

Unique constraint prevents duplicate values in specified columns, such as ensuring no two users can register with the same email address. Unique constraints are enforced at the database level for consistency.

Select for update is a SQL locking mechanism that provides pessimistic concurrency control by preventing other transactions from modifying selected records until the current transaction completes.

HTTP and Web Development Terms

REST API (Representational State Transfer) defines architectural principles for web services using standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources identified by URLs.

Middleware pattern processes HTTP requests before they reach route handlers, enabling cross-cutting concerns like authentication, logging, and error handling to be applied consistently across all endpoints.

Envelope pattern wraps API responses in metadata objects that include status information, pagination details, and error messages alongside the actual data payload.

Status codes are standardized HTTP response codes that indicate the outcome of requests: 200 for success, 400 for client errors, 401 for authentication failures, 500 for server errors.

System Design Patterns

Command Query Responsibility Segregation (CQRS) separates read operations from write operations, allowing each to be optimized independently. Product search queries can use read-optimized views while inventory updates use transactional write models.

Circuit breaker prevents cascading failures by monitoring downstream service health and failing fast when services become unavailable, with automatic recovery attempts after timeout periods.

Retry with exponential backoff handles transient failures by attempting failed operations multiple times with increasing delays between attempts, preventing overwhelming of recovering services.

Bulkhead pattern isolates system resources to prevent failures in one area from affecting others. Separate connection pools for read and write operations prevent long-running searches from blocking checkout transactions.

Business Domain Terms

Shopping cart represents a temporary collection of products that users intend to purchase, with operations for adding, removing, and updating quantities before proceeding to checkout.

Order lifecycle defines the stages an order passes through from creation to fulfillment: pending payment, payment authorized, payment captured, fulfillment in progress, shipped, delivered, completed.

Inventory management tracks product availability and prevents overselling through stock level monitoring and reservation systems that coordinate between browsing, cart management, and order processing.

Product catalog organizes and presents product information with search, filtering, and categorization capabilities that help customers discover and evaluate products.

Checkout process guides users through the final purchase steps including address collection, payment method selection, order review, and confirmation with appropriate validation at each stage.

User authentication verifies customer identity and maintains authenticated sessions that enable personalized experiences and secure access to account information and order history.

Implementation Guidance

Understanding these terms provides the foundation for implementing a robust e-commerce system. The terminology bridges business requirements with technical implementation, helping developers communicate effectively with stakeholders and make informed architectural decisions.

When implementing the system, refer to this glossary to ensure consistent use of terminology across code, documentation, and team communications. The precise definitions help avoid ambiguity in requirements and design discussions.

Key areas where terminology precision matters most include:

- Security implementations (understanding attack vectors and mitigation strategies)
- Transaction handling (ensuring data consistency and integrity)
- Performance optimization (identifying bottlenecks and scalability patterns)
- Error handling (distinguishing between operational errors and programming bugs)
- Testing strategies (selecting appropriate testing levels and isolation techniques)

The glossary serves as both a learning resource and a reference throughout the development process. As the system evolves and new features are added, the terminology provides a stable foundation for understanding how new concepts relate to existing system components.

For developers new to e-commerce systems, focusing on the business domain terms first provides essential context for understanding why certain technical patterns are necessary. The security and performance terms become more meaningful when understood in the context of actual business requirements for protecting customer data and providing responsive user experiences.