��# Complete E-Commerce Project Documentation > Project: Full-Stack E-Commerce Platform > Stack: Next.js 14 (App Router) + Supabase + Stripe + Resend + Vercel > Status: Production Ready (v1.0.0) > Live URL: https://store.shooshka.online --- ## Table of Contents 1. [System Architecture](#1-system-architecture) 2. [Full Code Review](#2-full-code-review) 3. [Core Concepts Deep Dive](#3-core-concepts-deep-dive) 4. [Problems & Solutions](#4-problems--solutions) 5. [Learning Summary](#5-learning-summary) 6. [Template Version](#6-template-version) --- # 1. System Architecture ## 1.1 Technology Stack Overview ### Frontend - Next.js 14 (App Router) - React framework with server-side rendering - React 18 - UI library with hooks and context - TypeScript - Type-safe JavaScript - Tailwind CSS - Utility-first CSS framework - Lucide React - Icon library ### Backend & Services - Next.js API Routes - Serverless backend endpoints (NOT NestJS - this is a Next.js-only project) - Supabase - PostgreSQL database + Authentication + Row Level Security - Stripe - Payment processing and webhooks - Resend - Transactional email service ### Infrastructure - Vercel - Hosting and deployment - Cloudflare - DNS management - Google OAuth - Social authentication ## 1.2 Architecture Diagram `⮝%%%�% �% USER BROWSER �% �% (React Components, Client-Side State, Context API) �% �%%%%%%%%%%%%%%%%%%%%%%,%%%�% �% �% HTTP Requests �% ⮝%%%�% �% NEXT.JS APP ROUTER (Vercel) �% �% ⮝%%%�% �% �% �% Server Components (SSR) �% �% �% �% - app/page.tsx �% �% �% �% - app/products/[id]/page.tsx �% �% �% �%%�% �% �% ⮝%%%�% �% �% �% Client Components ('use client') �% �% �% �% - components/AuthProvider.tsx �% �% �% �% - components/Navbar.tsx �% �% �% �% - app/auth/page.tsx �% �% �% �%%%�% �% �% ⮝%%�% �% �% �% API Routes (Route Handlers) �% �% �% �% - app/api/checkout/route.ts �% �% �% �% - app/api/webhook/route.ts �% �% �% �% - app/api/send-order-email/route.ts �% �% �% �%%%�% �% �%%%%%%%%%%%%%%,%%%%%%%%%%%%%%%%%%%%%%%%%%%%%,%%%%%%%%%%%%%%%%%%%%%%�% �% �% �% �% ⮝%%%%%%%%%�%%%%%%%%%%%�% ⮝%%%%%%%%%�%%%%%%%%%%%�% �% SUPABASE �% �% STRIPE �% �% - PostgreSQL �% �% - Payments �% �% - Auth �% �% - Webhooks �% �% - RLS �% �% - Checkout �% �%%%%%%%%%%%%%%%%%%%%%�% �%%%%%%%%%%%%%%%%%%%%%�% �% �% ⮝%%%%%%%%%�%%%%%%%%%%%�% �% RESEND �% �% - Emails �% �%%%%%%%%%%%%%%%%%%%%%�%` ## 1.3 Complete User Flow: "Add to Cart �! Order Delivered" ### Step-by-Step Flow #### Phase 1: Product Discovery 1. User visits homepage ( `app/page.tsx` ) - Server Component fetches products from Supabase - Displays products with `ProductGrid` component - Client-side search via `SearchBar` component 2. User clicks product �! `/products/[id]` - Server Component: `app/products/[id]/page.tsx` - Fetches product details from Supabase - Client Component: `ProductCard` for add-to-cart button #### Phase 2: Cart Management 3. User adds item to cart - Client Component calls `createSupabaseClient()` - Inserts into `cart_items` table: `typescript await supabase.from('cart_items').insert({ user_id: user.id, product_id: productId, quantity: 1 })` - RLS Policy: `auth.uid() = user_id` ensures user can only add to their cart 4. User views cart ( `app/cart/page.tsx` ) - Client Component fetches cart items - Joins with `products` table to get product details - Calculates total client-side #### Phase 3: Checkout 5. User clicks "Checkout" �! `/checkout` - Client Component: `app/checkout/page.tsx` - Collects shipping address (optional - can use Stripe's collection) - Calls API route: `POST /api/checkout` 6. API Route: `/api/checkout/route.ts` `typescript // Server-side Route Handler export async function POST(request: NextRequest) { // 1. Authenticate user (server-side) const supabase = createServerSupabaseClient() const { data: { user } } = await supabase.auth.getUser() // 2. Validate cart items const { items, address_id } = await request.json() // 3. Create Stripe Checkout Session const session = await stripe.checkout.sessions.create({ line_items: items.map(...), mode: 'payment', success_url:` ${NEXT_PUBLIC_APP_URL}/checkout/success?session_id={CHECKOUT_SESSION_ID} `, metadata: { user_id: user.id } }) // 4. Return session ID to client return NextResponse.json({ sessionId: session.id }) }` 7. Client redirects to Stripe Checkout - `window.location.href = session.url` - User completes payment on Stripe's secure page #### Phase 4: Payment

Processing 8. Stripe processes payment - Stripe validates card - Charges customer - Triggers webhook: `checkout.session.completed` 9. Webhook: `/api/webhook/route.ts` `typescript export async function POST(request: NextRequest) { // 1. Verify webhook signature (security) const signature = request.headers.get('stripe-signature') const event = stripe.webhooks.constructEvent(body, signature, STRIPE_WEBHOOK_SECRET) if (event.type === 'checkout.session.completed') { const session = event.data.object const userId = session.metadata.user_id // 2. Get cart items (using service role to bypass RLS) const supabaseAdmin = createClient(SUPABASE_URL, SERVICE_ROLE_KEY) const { data: cartItems } = await supabaseAdmin .from('cart_items') .select('*, products(*)') .eq('user_id', userId) // 3. Create order const { data: order } = await supabaseAdmin .from('orders') .insert({ user_id: userId, total: calculateTotal(cartItems), status: 'processing', stripe_payment_intent_id: session.id }) // 4. Create order items await supabaseAdmin.from('order_items').insert(cartItems.map(item => ({ order_id: order.id, product_id: item.product_id, quantity: item.quantity, price: item.products.price }))) // 5. Clear cart await supabaseAdmin.from('cart_items').delete().eq('user_id', userId) // 6. Remove purchased items from wishlist await supabaseAdmin.from('wishlist') .delete() .eq('user_id', userId) .in('product_id', cartItems.map(i => i.product_id)) // 7. Send order confirmation email const { data: authUser } = await supabaseAdmin.auth.admin.getUserById(userId) await sendOrderConfirmationEmail({ customerEmail: authUser.user.email, orderNumber: order.id.substring(0, 8).toUpperCase(), orderItems: cartItems.map(...), total: order.total }) } }` 10. User redirected to `/checkout/success?session_id=xxx` - Client Component: `app/checkout/success/page.tsx` - Checks if order exists (webhook may have already created it) - If not, waits 3 seconds, then creates order manually (fallback) - Clears cart if still present - Displays success message #### Phase 5: Order Fulfillment 11. Admin marks order as "shipped" (in Supabase Dashboard) - Updates `orders.status = 'shipped'` - Database trigger auto-generates `tracking_number` - Sets `shipped_at` timestamp 12. Admin triggers shipping email (via API or manual) - Calls `POST /api/send-shipping-email` - Sends email with tracking number via Resend 13. Order delivered - Admin updates `orders.status = 'delivered'` - Database trigger sets `delivered_at` timestamp - Admin triggers delivery email ## 1.4 Authentication Flow ### Google OAuth Flow (PKCE) `1. User clicks "Sign in with Google" �%%> Client: lib/auth/google.ts �%%> signInWithGoogle() �%%> supabase.auth.signInWithOAuth({ provider: 'google' }) �%%> Returns: { url: 'https://accounts.google.com/...' } 2. Browser redirects to Google �%%> User authenticates with Google �%%> Google redirects to: /auth/callback?code=xxx&state=xxx 3. Callback Route: app/auth/callback/route.ts �%%> Server-side Route Handler (GET) �%%> supabase.auth.exchangeCodeForSession(code) �%%> Supabase validates code + PKCE verifier �%%> Returns: { session: { access_token, refresh_token, user } } �%%> Sets cookies via setSession() �%%> Redirects to homepage 4. Client-side: AuthProvider.tsx �%%> useEffect listens to onAuthStateChange �%%> Updates user state �%%> All components re-render with new user` ### Email/Password Flow `1. User submits form (app/auth/page.tsx) �%%> Client Component �%%> supabase.auth.signInWithPassword({ email, password }) �%%> Supabase validates credentials �%%> Returns: { session, user } �%%> AuthProvider updates state �%%> Router redirects to homepage` ## 1.5 Data Flow: State Management ### Client-Side State - React Context: `AuthProvider` manages user authentication state - Local State: Each component uses `useState` for local UI state - Supabase Realtime: Not used in this project (could be added for live cart updates) ### Server-Side State - Database: Single source of truth (Supabase PostgreSQL) - RLS Policies: Enforce data access rules at database level - No Redux/Zustand: Simple Context API is sufficient for this app ### State Synchronization - Cart Count: Custom event `cartUpdated` dispatched, Navbar listens - User Auth: `onAuthStateChange` subscription in AuthProvider - Order Status: Polled on orders page (could use Supabase Realtime) --- # 2. Full Code Review ## 2.1 Project Structure `ecomm/ �%%% app/ # Next.js App Router �% �%%% api/ # API Routes (Route Handlers) �% �% �%%% checkout/ # Stripe checkout session creation �% �% �%%% webhook/ # Stripe webhook handler �% �% �%%% send-order-email/ # Order confirmation email �% �% �%%% send-shipping-email/ # Shipping notification email �% �% �%%% auth/ # Auth-related endpoints �% �%%% auth/ # Authentication pages �% �% �%%% page.tsx # Sign in/sign up form �% �% �%%% callback/ # OAuth callback handler �% �% �%%% reset-password/ # Password reset page �% �%%%`

`checkout/ # Checkout flow �% �% �%%% page.tsx # Checkout form �% �% �%%% success/ # Success page (order processing) �% �% �%%% cancel/ # Cancellation page �% �%%% products/ # Product pages �% �% �%%% [id]/ # Dynamic route for product detail �% �%%% cart/ # Shopping cart page �% �%%% orders/ # Order history page �% �%%% wishlist/ # Wishlist page �% �%%% profile/ # User profile page �% �%%% layout.tsx # Root layout (wraps all pages) �% �%%% page.tsx # Homepage �% �%%% globals.css # Global styles �% �%%% manifest.ts # PWA manifest �%%% components/ # React components �% �%%% AuthProvider.tsx # Auth context provider �% �%%% Navbar.tsx # Navigation bar �% �%%% ProductCard.tsx # Product display card �% �%%% ProductGrid.tsx # Grid of products �% �%%% SearchBar.tsx # Product search �% �%%% OrderTracking.tsx # Order status visualization �% �%%% ... �%%% lib/ # Utility libraries �% �%%% supabase/ # Supabase clients �% �% �%%% client.ts # Client-side Supabase �% �% �%%% server.ts # Server-side Supabase �% �%%% auth/ # Auth helpers �% �% �%%% google.ts # Google OAuth helper �% �%%% stripe.ts # Stripe client initialization �% �%%% email/ # Email utilities �% �%%% send.ts # Email sending functions �% �%%% templates/ # React Email templates �%%% hooks/ # Custom React hooks �% �%%% useWishlist.ts # Wishlist management hook �%%% types/ # TypeScript type definitions �% �%%% index.ts # All type interfaces �%%% middleware.ts # Next.js middleware �%%% next.config.js # Next.js configuration �%%% tailwind.config.ts # Tailwind CSS configuration �%%% package.json # Dependencies` ## 2.2 Critical Files Explained ### `app/layout.tsx` - Root Layout Purpose: Wraps all pages, provides global structure Key Concepts: - Server Component (default in App Router) - Metadata API: SEO and PWA configuration - Font Loading: Google Fonts (Poppins, Inter) with `next/font/google` - Context Provider: `AuthProvider` wraps entire app for global auth state Code Breakdown: `typescript // Server Component - runs on server export default function RootLayout({ children }) { return (<html> <body> <AuthProvider> {/* Client Component - provides auth context */} <Navbar /> {/* Client Component - navigation */} <main>{children}</main> {/* Page content */} </AuthProvider> </body> </html>) }` ### `components/AuthProvider.tsx` - Authentication Context Purpose: Manages user authentication state globally Key Concepts: - React Context API: `createContext` + `useContext` - useEffect: Side effects (session initialization, auth state listener) - Supabase Auth Listener: `onAuthStateChange` subscription - Client Component: Must be `'use client'` because it uses hooks Why This Pattern? - Single source of truth for user state - All components can access `user` via `useAuth()` hook - Automatic re-renders when auth state changes Flow: 1. Component mounts �! `useEffect` runs 2. Gets initial session: `supabase.auth.getSession()` 3. Sets up listener: `onAuthStateChange()` 4. When auth changes �! updates state �! all consumers re-render ### `lib/supabase/client.ts` - Client-Side Supabase Purpose: Creates Supabase client for browser use Key Concepts: - createClientComponentClient: Next.js helper for client-side Supabase - Automatic Cookie Handling: Manages auth cookies automatically - OAuth Support: Handles OAuth redirects and PKCE Why This Helper? - Next.js-specific optimizations - Proper cookie handling for SSR/hydration - Built-in OAuth flow support ### `lib/supabase/server.ts` - Server-Side Supabase Purpose: Creates Supabase client for server components/API routes Key Concepts: - createServerComponentClient: For Server Components - createRouteHandlerClient: For API Routes (used in webhook) - Cookie Access: Reads cookies from `next/headers` When to Use Which? - Server Components �! `createServerComponentClient` - API Routes �! `createRouteHandlerClient` - Client Components �! `createClientComponentClient` (from client.ts) ### `app/api/webhook/route.ts` - Stripe Webhook Handler Purpose: Processes Stripe payment events server-side Key Concepts: - Route Handler: `export async function POST()` - Webhook Signature Verification: Security - ensures request is from Stripe - Service Role Key: Bypasses RLS to create orders - Idempotency: Checks for existing orders to prevent duplicates Security Flow: 1. Stripe sends POST request with signature header 2. Verify signature: `stripe.webhooks.constructEvent()` 3. Process event (only if signature valid) 4. Return 200 OK (Stripe retries on non-200) Why Service Role Key? - Webhook runs server-side, no user session - Need to create orders for any user - RLS would block without `auth.uid()` context ### `app/checkout/success/page.tsx` - Order Processing Fallback Purpose: Handles order creation if webhook fails Key Concepts: - useRef: Prevents duplicate processing on re-renders - Race Condition Handling: Waits for webhook, then creates order manually - localStorage: Prevents duplicate email sends on refresh Why This Fallback? - Webhooks can fail (network issues, timeouts) - User already paid, must create order - Dual-write pattern: webhook (primary) + success page (fallback) ### `lib/auth/google.ts` - Google OAuth Helper Purpose: Client-side helper to initiate Google

OAuth flow Key Concepts: - PKCE Flow: Proof Key for Code Exchange (security) - Redirect URL: Must match Supabase/Google Console config - Environment-Aware: Uses `NEXT_PUBLIC_APP_URL` for production OAuth Flow: 1. User clicks button �! `signInWithGoogle()` called 2. Supabase generates PKCE code verifier + challenge 3. Redirects to Google with `code_challenge` 4. User authenticates �! Google redirects to `/auth/callback?code=xxx` 5. Callback exchanges code for session PKCE Error Handling: - If code verifier missing (incognito mode) �! auto-retry - Redirects to `/auth?oauth_retry=true` - Auth page automatically retries OAuth flow ### `lib/email/send.ts` - Email Sending Purpose: Sends transactional emails via Resend Key Concepts: - React Email: JSX-based email templates - Server-Side Only: Must run in API routes (has env vars) - Template Rendering: `render()` converts React component to HTML Email Types: 1. Order Confirmation: Sent after payment (webhook or fallback) 2. Shipping Notification: Sent when order marked "shipped" 3. Delivery Notification: Sent when order marked "delivered" Why React Email? - Type-safe email templates - Component reusability - Easy to maintain and test --- # 3. Core Concepts Deep Dive ## 3.1 Next.js App Router Concepts ### App Router vs Pages Router This project uses App Router (Next.js 13+) Key Differences: - Routing: File-based routing in `app/` directory - Server Components: Default (no `'use client'` needed) - Layouts: Nested layouts with `layout.tsx` - Loading States: Built-in `loading.tsx` - Error Handling: Built-in `error.tsx` ### Server Components vs Client Components Server Components (Default): ` typescript // app/page.tsx - Server Component export default async function HomePage() { // Can directly access database, no 'use client' const supabase = createServerSupabaseClient() const { data: products } = await supabase.from('products').select('*') return <ProductGrid products={products} /> } ` Benefits: - Runs on server (faster, no JS bundle) - Direct database access - Secure (API keys never exposed) - Better SEO (HTML rendered server-side) Client Components (Explicit): ` typescript // components/Navbar.tsx 'use client' // Must declare export default function Navbar() { const [isOpen, setIsOpen] = useState(false) // Needs client-side state // ... } ` When to Use Client Components: - Interactive UI (buttons, forms, state) - Browser APIs (localStorage, window) - Event handlers (onClick, onChange) - React hooks (useState, useEffect, useContext) ### Routing & Route Groups File-Based Routing: ` app/ page.tsx �! / products/ page.tsx �! /products [id]/ page.tsx �! /products/[id] (dynamic) checkout/ success/ page.tsx �! /checkout/success ` Dynamic Routes: - `[id]` - Single dynamic segment - `[...slug]` - Catch-all routes - `(group)` - Route groups (organizational, don't affect URL) Route Handlers (API Routes): ` app/api/ checkout/ route.ts �! POST /api/checkout webhook/ route.ts �! POST /api/webhook ` ### Layouts & Nested Layouts Root Layout ( `app/layout.tsx` ): - Wraps ALL pages - Contains `<html>` , `<body>` - Global providers (AuthProvider) - Global UI (Navbar, Footer) Nested Layouts: ` app/ layout.tsx # Root layout dashboard/ layout.tsx # Dashboard-specific layout page.tsx # Uses both layouts ` Layout Pattern: - Shared UI (headers, sidebars) - Persistent state across navigation - Loading states per route group ### Navigation Link Component: ` typescript import Link from 'next/link' <Link href="/products/123">Product</Link> ` useRouter Hook: ` typescript 'use client' import { useRouter } from 'next/navigation' const router = useRouter() router.push('/products/123') router.replace('/products/123') // No history entry router.refresh() // Re-fetch server components ` Why `next/navigation` not `next/router` ? - App Router uses new navigation API - Pages Router uses old `next/router` ### Context API and Providers Pattern Used in This Project: ` typescript // 1. Create Context const AuthContext = createContext<AuthContextType>({ user: null, loading: true, signOut: async () => {} }) // 2. Create Provider Component export function AuthProvider({ children }) { const [user, setUser] = useState(null) // ... auth logic return (<AuthContext.Provider value={{ user, loading, signOut }}> {children} </AuthContext.Provider>) } // 3. Create Hook for Easy Access export const useAuth = () => useContext(AuthContext) // 4. Use in Components function MyComponent() { const { user, loading } = useAuth() // ... } ` Why Context Over Redux? - Simpler for small-to-medium apps - Built into React (no extra dependency) - Sufficient for auth state (not complex state) ### React Hooks Deep Dive useState: ` typescript const [count, setCount] = useState(0) // State persists across re-renders // Triggers re-render when updated ` useEffect: ` typescript useEffect(() => { // Side effect (API call, subscription) const subscription = supabase.auth.onAuthStateChange(...) return () => { // Cleanup (unsubscribe) subscription.unsubscribe() } }, [dependencies]) // Run when dependencies change ` useCallback: ` typescript const fetchData = useCallback(async () => { // Expensive function }, [dependency]) //

Memoize function, recreate only if dependency changes useMemo: typescript const filteredProducts = useMemo(() => { return products.filter(p => p.name.includes(searchTerm)) }, [products, searchTerm]) // Memoize result, recalculate only if inputs change useRef: typescript const processedRef = useRef(false) // Persists across re-renders, doesn't trigger re-render if (!processedRef.current) { processedRef.current = true // Do something once } ### Server Actions (Not Used in This Project) What They Are: - Functions that run on server - Can be called from Client Components - Alternative to API routes Why Not Used? - API routes are more explicit - Better for webhooks (external services) - More control over request/response ### API Routes (Route Handlers) Pattern: typescript // app/api/checkout/route.ts import { NextRequest, NextResponse } from 'next/server' export async function POST(request: NextRequest) { const body = await request.json() // Process request return NextResponse.json({ success: true }) } HTTP Methods: - GET , POST , PUT , DELETE , PATCH - Export function with method name When to Use: - External API integration (Stripe, Resend) - Webhooks (Stripe webhook handler) - Server-side operations (email sending) ### Cache & Revalidation Default Caching: - Server Components: Cached by default - API Routes: Not cached by default Force Revalidation: typescript export const revalidate = 0 // No cache export const dynamic = 'force-dynamic' // Always dynamic This Project: - Most pages use default caching - API routes are dynamic (webhooks, checkout) ### Hydration & Hydration Errors What is Hydration? - Server renders HTML - Client "hydrates" with React - React attaches event listeners Common Errors: - Server HTML " Client HTML - localStorage used in Server Component - Date/time mismatches **Fixes in This Project**: - Check typeof window !== 'undefined' before browser APIs - Use useEffect for client-only code - Avoid localStorage in initial render ### Middleware **File**: middleware.ts **Current Implementation**: typescript export async function middleware(req: NextRequest) { // Currently minimal - just passes through return NextResponse.next() } **Could Be Used For**: - Route protection (redirect if not authenticated) - A/B testing - Geolocation-based routing - Request logging **Why Not Used for Auth?** - Client-side protection in components is sufficient - More flexible (can show different UI for unauthenticated users) ### Authentication Flow (Detailed) **1. Initial Load**: Browser �! Next.js Server �! Renders HTML (no user data) �! Sends HTML to browser �! Browser hydrates React �! AuthProvider useEffect runs �! Gets session from Supabase �! Updates state �! Components re-render with user **2. Sign In**: User submits form �! Client Component calls supabase.auth.signInWithPassword() �! Supabase validates �! Returns session �! AuthProvider onAuthStateChange fires �! Updates user state �! Router redirects to homepage **3. OAuth Sign In**: User clicks "Sign in with Google" �! signInWithGoogle() called �! Redirects to Google �! User authenticates �! Google redirects to /auth/callback?code=xxx �! Callback route exchanges code for session �! Sets cookies �! Redirects to homepage �! AuthProvider detects session �! Updates state ### Environment Variables **Next.js Env Var Rules**: - NEXT_PUBLIC_ *: Exposed to browser - Others: Server-only (API routes, Server Components) **This Project Uses**: env # Public (browser-accessible) NEXT_PUBLIC_SUPABASE_URL=... NEXT_PUBLIC_SUPABASE_ANON_KEY=... NEXT_PUBLIC_STRIPE_PUBLISHABLE_KEY=... NEXT_PUBLIC_APP_URL=https://store.shooshka.online # Server-only (secure) SUPABASE_SERVICE_ROLE_KEY=... # Never expose! STRIPE_SECRET_KEY=... STRIPE_WEBHOOK_SECRET=... RESEND_API_KEY=... RESEND_FROM_EMAIL=... **Why Service Role Key is Server-Only**: - Bypasses RLS (can access any user's data) - Must never be exposed to browser - Only used in webhook (server-side) ### ISR / SSR / SSG **SSR (Server-Side Rendering)** - Used in This Project: typescript // app/page.tsx export default async function HomePage() { const products = await fetchProducts() // Runs on every request return <ProductGrid products={products} /> } **SSG (Static Site Generation)** - Not Used: - Would require generateStaticParams() for dynamic routes - Not suitable for dynamic product data **ISR (Incremental Static Regeneration)** - Not Used: - Would require revalidate export - Not needed for this use case **Why SSR?** - Product data changes frequently - User-specific data (cart, orders) - Real-time updates needed ## 3.2 Supabase Concepts ### Row Level Security (RLS) **What is RLS?** - Database-level security - Policies control who can read/write data - Enforced at PostgreSQL level (not application level) **Example Policy**: sql -- Users can only see their own cart items CREATE POLICY "Users can view their own cart" ON cart_items FOR SELECT USING (auth.uid() = user_id); **How It Works**: 1. User makes request �! Supabase checks auth.uid() 2.*

Applies matching policies 3. Returns only allowed rows **Why RLS?** - Security at database level (can't bypass) - No need to check permissions in every query - Works even if application code has bugs ### Policies **Types of Policies**: - **SELECT**: Who can read - **INSERT**: Who can create - **UPDATE**: Who can modify - **DELETE**: Who can remove **Policy Functions**: - auth.uid() : Current user's ID - auth.role() : User's role - auth.email() : User's email **This Project's Policies**: sql -- Products: Public read, no write CREATE POLICY "Products are viewable by everyone" ON products FOR SELECT USING (true); -- Cart: Users can only access their own CREATE POLICY "Users can view their own cart" ON cart_items FOR SELECT USING (auth.uid() = user_id); -- Orders: Users can only see their own CREATE POLICY "Users can view their own orders" ON orders FOR SELECT USING (auth.uid() = user_id); ### Auth Strategies **Email/Password**: - Traditional sign-up/sign-in - Password hashed by Supabase - Email confirmation optional **OAuth (Google)**: - Social authentication - PKCE flow for security - No password needed **This Project Uses Both**: - Email/password for traditional users - Google OAuth for convenience ### Postgres Schema Structure **Tables**: 1. **products**: Product catalog 2. **cart_items**: Shopping cart (user_id + product_id) 3. **orders**: Order headers (user_id + total + status) 4. **order_items**: Order line items (order_id + product_id) 5. **wishlist**: User wishlists 6. **profiles**: User profile data 7. **user_addresses**: Shipping addresses **Relationships**: - cart_items.user_id �! auth.users.id - cart_items.product_id �! products.id - orders.user_id �! auth.users.id - order_items.order_id �! orders.id - order_items.product_id �! products.id **Foreign Keys**: - Ensure data integrity - Cascade deletes (if user deleted, cart deleted) - Prevent orphaned records ### Realtime Changes (Not Used, But Available) **Supabase Realtime**: - WebSocket connection - Listen to database changes - Could be used for live cart updates **Example (Not Implemented)**: typescript supabase .channel('cart-changes') .on('postgres_changes', { event: 'INSERT', schema: 'public', table: 'cart_items', filter: user_id=eq.${user.id} }, (payload) => { // Update cart UI }) .subscribe() ### Linking User Metadata **auth.users Table**: - Managed by Supabase - Contains: id, email, created_at, etc. **profiles Table**: - Custom user data - Foreign key: id �! auth.users.id - Contains: full_name, phone, etc. **Accessing User Data**: typescript // Get auth user const { data: { user } } = await supabase.auth.getUser() // Get profile const { data: profile } = await supabase .from('profiles') .select('') .eq('id', user.id) .single() ### Webhooks Integration **Supabase Webhooks** (Not Used in This Project): - Database triggers can call webhooks - Could notify external services on data changes **This Project Uses**: - Stripe webhooks (payment events) - Not Supabase webhooks ### Filtering / Ordering Data **Supabase Query Builder**: typescript const { data } = await supabase .from('products') .select('') .eq('category', 'electronics') // Filter .order('price', { ascending: false }) // Sort .limit(10) // Paginate **Common Methods**: - .eq() : Equal - .neq() : Not equal - .gt() : Greater than - .lt() : Less than - .like() : Pattern match - .in() : In array - .order() : Sort - .limit() : Paginate ### Handling Order Status Updates **Database Triggers**: sql -- Auto-generate tracking number when status = 'shipped' CREATE TRIGGER update_order_status_trigger BEFORE UPDATE ON orders FOR EACH ROW EXECUTE FUNCTION update_order_status_and_tracking(); **Trigger Function**: sql CREATE FUNCTION update_order_status_and_tracking() RETURNS TRIGGER AS $$ BEGIN -- Set shipped_at when status changes to 'shipped' IF NEW.status = 'shipped' AND OLD.status != 'shipped' THEN NEW.shipped_at = NOW(); -- Auto-generate tracking number if not set IF NEW.tracking_number IS NULL THEN NEW.tracking_number = generate_tracking_number(); END IF; END IF; -- Set delivered_at when status changes to 'delivered' IF NEW.status = 'delivered' AND OLD.status != 'delivered' THEN NEW.delivered_at = NOW(); END IF; RETURN NEW; END; $$ LANGUAGE plpgsql; **Why Triggers?** - Automatic timestamp updates - Data consistency - No need to remember to set fields in application code ## 3.3 Stripe Concepts ### Payment Intent Flow (Not Used - Uses Checkout Sessions Instead) **This Project Uses Checkout Sessions** (simpler): - Stripe-hosted checkout page - No need to build payment form - Handles all payment methods ### Client Secret Usage (Not Used) **This Project Uses**: - Checkout Sessions (redirect to Stripe) - Not Payment Intents (embedded form) **Why Checkout Sessions?** - Simpler implementation - Better UX (Stripe's optimized checkout) - Handles all edge cases ### Webhook Signature Verification **Critical Security**: typescript const signature = request.headers.get('stripe-signature') const event = stripe.webhooks.constructEvent(body, signature, process.env.STRIPE_WEBHOOK_SECRET) **Why Verify?** - Prevents fake webhooks - Ensures request is from Stripe - Protects against replay attacks **How It Works**: 1. Stripe signs request with secret 2. Server verifies signature 3. Only processes if valid ### Order Creation Flow **Sequence**: 1. User clicks checkout �! API creates session 2. User pays on Stripe �! Stripe processes payment 3. Stripe sends webhook �! Server creates order 4. User redirected �! Success page shows order **Idempotency**: - Check for existing order before creating - Prevents duplicates if webhook fires twice - Uses stripe_payment_intent_id as unique key ### Handling Cart Clearing State **Problem**: Cart must be cleared after payment, but when? **Solution**: Dual approach 1. **Webhook clears cart** (primary) 2. **Success page clears cart** (fallback) **Why Both?** - Webhook might fail (network, timeout) - User already paid, must clear cart - Success page ensures cart is cleared ### Stripe Portal vs Session **Checkout Session** (Used): - One-time payment - Redirects to Stripe - Returns to success URL **Customer Portal** (Not Used): - Subscription management - Update payment methods - View invoices **This Project**: One-time payments only �! Checkout Sessions ### Idempotency Keys **Not Used in This Project** (but recommended): typescript // Stripe supports idempotency keys await stripe.checkout.sessions.create({ // ... }, { idempotencyKey: 'unique-key-per-request' }) **Why Use?** - Prevents duplicate charges - Retry-safe - Better for production **This Project Uses**: - Database-level idempotency (check for existing order) - Simpler, but less robust ## 3.4 Resend Concepts ### Email Templates **React Email**: - JSX-based templates - Type-safe - Component-based **Example**: typescript // lib/email/templates/OrderConfirmation.tsx export default function OrderConfirmationEmail({ orderNumber, customerName, ... }) { return (<Html> <Head /> <Body> <Container> <Heading>Order Confirmed!</Heading> <Text>Hi {customerName},</Text> <Text>Your order #{orderNumber} has been confirmed.</Text> </Container> </Body> </Html>) } **Why React Email?** - Familiar syntax (JSX) - Type-safe props - Easy to maintain - Can use React components ### Triggering Emails Server-Side **Must Be Server-Side**: - API keys must be secret - Cannot expose in browser - Must run in API routes or Server Components **Pattern**: typescript // app/api/send-order-email/route.ts export async function POST(request: NextRequest) { const { orderId, userId } = await request.json() // Get order data const order = await getOrder(orderId) // Send email await sendOrderConfirmationEmail({ customerEmail: order.user.email, orderNumber: order.id, // ... }) return NextResponse.json({ success: true }) } ### Order Confirmation Email Flow **When Sent**: 1. **Webhook** (primary): After payment processed 2. **Success Page** (fallback): If webhook didn't fire **Data Needed**: - Customer email - Order number - Order items - Total amount - Order date **Template**: lib/email/templates/OrderConfirmation.tsx ### Shipping Update Flow **When Sent**: - Admin marks order as "shipped" - Admin triggers email (via API or manual) **Data Needed**: - Customer email - Order number - Tracking number - Estimated delivery date - Order URL (for "View Order" button) **Template**: lib/email/templates/ShippingNotification.tsx ### Redirecting User to "View Order Details" **Email Button**: typescript <Button href={ ${orderUrl}/orders?orderId=${orderId} }> View Your Order Details </Button> **Orders Page**: - Reads orderId from URL query - Auto-expands that order - Scrolls to order **Implementation**: typescript // app/orders/page.tsx const searchParams = useSearchParams() const orderId = searchParams.get('orderId') useEffect(() => { if (orderId) { setExpandedOrder(orderId) // Scroll to order } }, [orderId]) ### Best Practices for Email Reliability **1. Error Handling**: typescript try { await sendEmail(...) } catch (error) { // Log error, don't fail request console.error('Email failed:', error) // Order still created, email can be retried } **2. Environment Checks**: typescript if (!process.env.RESEND_API_KEY) { return { success: false, error: 'API key not configured' } } **3. Domain Verification**: - Verify domain in Resend dashboard - Use verified domain in RESEND_FROM_EMAIL - Prevents "Not authorized" errors **4. Retry Logic** (Not Implemented, But Recommended): - Queue failed emails - Retry with exponential backoff - Use job queue (Bull, BullMQ) --- # 4. Problems We Struggled With & Their Correct Solutions ## 4.1 Google OAuth Redirect Loop **Problem**: - User signs in with Google - Gets stuck in redirect loop - Never reaches homepage **Root Cause**: - Incorrect redirect URL configuration - Mismatch between Supabase and Google Console - PKCE code verifier lost (incognito mode) **Early Attempts** (Wrong): - Hardcoded localhost URLs - Missing NEXT_PUBLIC_APP_URL env var - Not handling PKCE errors **Correct Solution**: typescript // lib/auth/google.ts const origin = process.env.NEXT_PUBLIC_APP_URL || window.location.origin const callbackUrl = ${origin}/auth/callback?

next=${encodeURIComponent(redirectTo)} // app/auth/callback/route.ts if (exchangeError.message?.includes('code verifier')) { // Auto-retry OAuth flow (silent) return NextResponse.redirect(new URL('/auth?oauth_retry=true&provider=google', url.origin)) } **Key Learnings**: - Always use environment-aware URLs - Handle PKCE errors gracefully - Auto-retry for incognito mode ## 4.2 Stripe Webhook Failures **Problem**: - Webhook not firing - Orders not created after payment - Cart not cleared **Root Cause**: - Webhook URL not configured in Stripe - Signature verification failing - Local development (Stripe can't reach localhost) **Early Attempts** (Wrong): - Testing webhooks locally (impossible) - Not verifying signatures - Not handling webhook retries **Correct Solution**: 1. **Use Stripe CLI for local testing**: bash stripe listen --forward-to localhost:3000/api/webhook 2. **Verify signatures**: typescript const event = stripe.webhooks.constructEvent(body, signature, STRIPE_WEBHOOK_SECRET) 3. **Idempotency**: typescript // Check for existing order const { data: existingOrder } = await supabaseAdmin .from('orders') .select('id') .eq('stripe_payment_intent_id', sessionId) .single() if (existingOrder) { return NextResponse.json({ received: true }) } 4. **Fallback in success page**: - Wait 3 seconds for webhook - If no order, create manually **Key Learnings**: - Always verify webhook signatures - Implement idempotency - Have fallback mechanisms ## 4.3 Supabase RLS Blocking Updates **Problem**: - Webhook can't create orders - RLS policies blocking admin operations **Root Cause**: - Webhook has no user context - RLS checks auth.uid() which is null - Policies block all operations **Early Attempts** (Wrong): - Trying to bypass RLS with anon key - Not understanding RLS behavior **Correct Solution**: typescript // Use service role key (bypasses RLS) const supabaseAdmin = createClient(process.env.NEXT_PUBLIC_SUPABASE_URL!, process.env.SUPABASE_SERVICE_ROLE_KEY! // Not anon key!) **Why Service Role Key?** - Bypasses all RLS policies - Can access any user's data - Required for webhooks (no user context) **Security Note**: - Service role key is SECRET - Never expose to browser - Only use server-side ## 4.4 Wishlist Not Clearing After Purchase **Problem**: - Items remain in wishlist after purchase - User sees purchased items in wishlist **Root Cause**: - Not removing wishlist items in webhook - Only clearing cart, not wishlist **Early Attempts** (Wrong): - Trying to clear wishlist client-side - Not handling in webhook **Correct Solution**: typescript // In webhook, after creating order const purchasedProductIds = cartItems.map(item => item.product_id) if (purchasedProductIds.length > 0) { await supabaseAdmin .from('wishlist') .delete() .eq('user_id', userId) .in('product_id', purchasedProductIds) } **Key Learnings**: - Clear wishlist in webhook (server-side) - Use .in() for bulk delete - Handle edge cases (empty array) ## 4.5 Cart Not Clearing After Payment **Problem**: - Cart still has items after successful payment - User sees old items in cart **Root Cause**: - Race condition between webhook and success page - Webhook might fail, cart not cleared **Early Attempts** (Wrong): - Only clearing in webhook - Not handling failures **Correct Solution**: **Dual-write pattern**: 1. Webhook clears cart (primary) 2. Success page clears cart (fallback) typescript // In success page const { data: cartItems } = await supabase .from('cart_items') .select('') .eq('user_id', user.id) if (cartItems && cartItems.length > 0) { // Webhook didn't clear, clear now await supabase .from('cart_items') .delete() .eq('user_id', user.id) } **Key Learnings**: - Always have fallback mechanisms - Check state before assuming - Handle race conditions ## 4.6 Order ID Mismatch **Problem**: - Duplicate orders created - Same payment creates multiple orders **Root Cause**: - Webhook fires multiple times - Success page creates order even if webhook did - No idempotency check **Early Attempts** (Wrong): - Not checking for existing orders - Creating orders without validation **Correct Solution**: typescript // In webhook const { data: existingOrder } = await supabaseAdmin .from('orders') .select('id') .eq('stripe_payment_intent_id', sessionId) .single() if (existingOrder) { return NextResponse.json({ received: true }) } // In success page const { data: existingOrder } = await supabase .from('orders') .select(' ') .eq('stripe_payment_intent_id', sessionId) .single() if (existingOrder) { // Order exists, skip creation return } **Key Learnings**: - Always check for existing records - Use unique identifiers (session ID) - Implement idempotency ## 4.7 DNS & Cloudflare Proxy Problems **Problem**: - Vercel shows "Proxy Detected" warning - Domain not working correctly **Root Cause**: - Cloudflare proxy enabled (orange cloud) - Vercel can't verify domain ownership - DNS conflicts **Early Attempts** (Wrong): - Not understanding Cloudflare proxy - Incorrect DNS records **Correct Solution**: 1. **Disable Cloudflare proxy** (gray cloud): -

CNAME record: store.shooshka.online �! cname.vercel-dns.com - Proxy status: DNS only (gray cloud) 2. **Wait for propagation**: - DNS changes take time - Vercel needs to verify 3. **Verify in Vercel**: - Check domain configuration - Should show "Valid Configuration" **Key Learnings**: - Understand DNS proxy behavior - Vercel needs direct DNS access - Be patient with DNS propagation ## 4.8 Vercel Domain Configuration **Problem**: - OAuth redirects to wrong URL - Emails have localhost links **Root Cause**: - NEXT_PUBLIC_APP_URL not set in Vercel - Using window.location.origin (can be wrong) **Early Attempts** (Wrong): - Hardcoded URLs - Not using environment variables **Correct Solution**: typescript // Always use environment variable const origin = process.env.NEXT_PUBLIC_APP_URL || window.location.origin // In production, force production URL const isProduction = process.env.NODE_ENV === 'production' const finalUrl = (origin.includes('localhost') && isProduction) ? 'https://store.shooshka.online' : origin **Key Learnings**: - Always use env vars for URLs - Handle localhost in production - Test with production URLs ## 4.9 Protected Routes Based on User Session **Problem**: - Users accessing protected pages without auth - No redirect to login **Root Cause**: - Not checking auth in components - Relying only on RLS (not enough) **Early Attempts** (Wrong): - Trying to protect in middleware (too complex) - Not checking auth state **Correct Solution**: typescript // In protected page component 'use client' export default function OrdersPage() { const { user, loading } = useAuth() const router = useRouter() useEffect(() => { if (!loading && !user) { router.push('/auth') } }, [user, loading, router]) if (loading) return <Loading /> if (!user) return null // Render protected content } **Key Learnings**: - Client-side protection is simpler - Use Context for auth state - Show loading states ## 4.10 Context Not Updating After Authentication **Problem**: - User signs in, but UI doesn't update - Components still show "Sign In" button **Root Cause**: - Context not listening to auth changes - Not subscribing to onAuthStateChange **Early Attempts** (Wrong): - Only checking session on mount - Not listening to changes **Correct Solution**: typescript // In AuthProvider useEffect(() => { const supabase = createSupabaseClient() // Get initial session supabase.auth.getSession().then(({ data: { session } }) => { setUser(session?.user ?? null) }) // Listen for auth changes const { data: { subscription } } = supabase.auth.onAuthStateChange((event, session) => { setUser(session?.user ?? null) setLoading(false) }) return () => subscription.unsubscribe() }, []) **Key Learnings**: - Always subscribe to auth changes - Handle initial session - Clean up subscriptions ## 4.11 Missing Environment Variables **Problem**: - Emails not sending - Stripe not working - OAuth failing **Root Cause**: - Environment variables not set in Vercel - Using defaults (wrong values) **Early Attempts** (Wrong): - Not checking for env vars - Using hardcoded values **Correct Solution**: typescript // Always check for required env vars if (!process.env.RESEND_API_KEY) { console.error('RESEND_API_KEY not set!') return { success: false, error: 'API key not configured' } } // Provide helpful error messages if (!process.env.NEXT_PUBLIC_APP_URL) { console.warn('NEXT_PUBLIC_APP_URL not set, using window.location.origin') } **Key Learnings**: - Always validate env vars - Provide helpful error messages - Document required variables ## 4.12 Route Groups Not Rendering Properly **Problem**: - Pages not rendering - Layouts not applying **Root Cause**: - Incorrect file structure - Missing page.tsx files **Early Attempts** (Wrong): - Not understanding App Router structure - Missing required files **Correct Solution**: app/ layout.tsx # Required: Root layout page.tsx # Required: Homepage products/ [id]/ page.tsx # Required: Product page **Key Learnings**: - App Router requires page.tsx for routes - layout.tsx wraps child routes - Understand file-based routing ## 4.13 App Router Caching Interfering with Dynamic Data **Problem**: - Product data not updating - Stale data shown **Root Cause**: - Server Components cached by default - Not revalidating **Early Attempts** (Wrong): - Not understanding caching - Trying to force client-side **Correct Solution**: typescript // Force dynamic rendering export const dynamic = 'force-dynamic' export const revalidate = 0 // Or use cache: 'no-store' in fetch const { data } = await supabase .from('products') .select('') // Supabase client handles caching **Key Learnings**: - Understand Next.js caching - Use revalidate when needed - Cache is good for performance, but know when to disable --- # 5. Learning Summary ## 5.1 Important React Patterns ### 1. Context API for Global State **When to Use**: - Auth state (user, loading) - Theme preferences - Simple global state **When NOT to Use**: - Complex state (use Redux/Zustand) - Frequently changing data (use React Query) - Large datasets (use database) ### 2. Custom Hooks for Reusability **Pattern**: typescript // hooks/useWishlist.ts export function useWishlist(userEmail: string | null) { const [items, setItems] = useState([]) // ... logic return { items, addToWishlist, removeFromWishlist } } // Usage const { items, addToWishlist } = useWishlist(user?.email) **Benefits**: - Reusable logic - Clean component code - Easy to test ### 3. useEffect for Side Effects **Always Clean Up**: typescript useEffect(() => { const subscription = supabase.auth.onAuthStateChange(...) return () => subscription.unsubscribe() // Cleanup! }, []) **Dependency Array**: - Empty [] : Run once on mount - [dep] : Run when dependency changes - No array: Run on every render (usually wrong!) ### 4. useRef for Persistent Values **Use Cases**: - Prevent duplicate operations - Store previous values - Access DOM elements **Example**: typescript const processedRef = useRef(false) if (!processedRef.current) { processedRef.current = true // Do something once } ### 5. Conditional Rendering Patterns typescript // Early return if (loading) return <Loading /> if (!user) return null // Ternary {user ? <UserMenu /> : <SignInButton />} // Logical AND {user && <ProtectedContent />} ## 5.2 Important Next.js App Router Concepts ### 1. Server Components by Default **Remember**: Components are Server Components unless marked 'use client' **Benefits**: - Smaller bundle size - Better performance - Direct database access ### 2. File-Based Routing **Convention**: File structure = URL structure app/products/[id]/page.tsx �! /products/[id] ### 3. Route Handlers for API Endpoints **Pattern**: app/api/[route]/route.ts typescript export async function POST(request: NextRequest) { // Handle POST request } ### 4. Layouts for Shared UI **Pattern**: layout.tsx wraps child routes - Root layout wraps everything - Nested layouts wrap specific sections ### 5. Metadata API typescript export const metadata: Metadata = { title: 'Page Title', description: 'Page description' } ## 5.3 Authentication Architecture ### Key Principles 1. **Server-Side Session Management** - Supabase handles sessions - Cookies managed automatically - No manual token storage 2. **Client-Side State Sync** - Context API for global state - onAuthStateChange for updates - Automatic re-renders 3. **Protected Routes** - Client-side checks (simpler) - Redirect if not authenticated - Show loading states 4. **OAuth Flow** - PKCE for security - Handle errors gracefully - Auto-retry for edge cases ## 5.4 How to Design a Full-Stack SaaS ### Architecture Decisions 1. **Choose Your Stack** - Frontend: Next.js (React) - Backend: Next.js API Routes (or separate) - Database: Supabase (PostgreSQL) - Auth: Supabase Auth - Payments: Stripe - Emails: Resend 2. **State Management** - Global: Context API (simple) or Redux (complex) - Server: Database (single source of truth) - Local: useState (component state) 3. **Security** - RLS policies (database level) - Environment variables (secrets) - Webhook signature verification - Input validation 4. **Error Handling** - Try-catch blocks - Fallback mechanisms - User-friendly error messages - Logging for debugging 5. **Performance** - Server Components (reduce bundle) - Image optimization - Caching strategies - Database indexing ## 5.5 Debugging Mindset ### 1. Read Error Messages Carefully - TypeScript errors are helpful - Stack traces show exact location - Error messages often suggest fixes ### 2. Use Console Logging Strategically typescript console.log('=� � [Component] State:', state) console.error('L' [API] Error:', error) ### 3. Check Network Tab - API requests/responses - Webhook deliveries - CORS issues ### 4. Verify Environment Variables - Check Vercel dashboard - Test locally with .env.local - Use different values for dev/prod ### 5. Test Incrementally - Test each feature in isolation - Don't change multiple things at once - Use version control (git commits) ### 6. Understand the Flow - Trace data flow - Check each step - Verify assumptions ## 5.6 Production Deployment Lessons ### 1. Environment Variables - Set in Vercel dashboard - Use different values for dev/prod - Never commit secrets ### 2. Domain Configuration - DNS propagation takes time - Verify domain ownership - Test with production URLs ### 3. Webhook Configuration - Use Stripe CLI for local testing - Configure webhook URL in Stripe - Verify signatures ### 4. Email Configuration - Verify domain in Resend - Use verified domain in FROM_EMAIL - Test email delivery ### 5. Database Migrations - Test migrations locally first - Backup before migrating - Use transactions when possible ### 6. Monitoring - Check Vercel logs - Monitor Stripe webhooks - Track email delivery ## 5.7 Mistakes to Avoid Next Time ### 1. Don't Hardcode URLs L' const url = 'http://localhost:3000' ⍰' const url = process.env.NEXT_PUBLIC_APP_URL ### 2. Don't Skip Error Handling L' await supabase.from('orders').insert(data) �' typescript const { data, error } = await supabase.from('orders').insert(data) if (error) { console.error('Failed to create order:', error) return { success: false, error: error.message } } `### 3. Don't Ignore TypeScript Errors L' Ignoring type errors �' Fix type errors immediately ### 4. Don't Test Only Locally L' Only testing on localhost �' Test with production URLs, test on Vercel ### 5. Don't Forget Idempotency L' Creating orders without checking �' Always check for existing records ### 6. Don't Expose Secrets L' Using service role key in client �' Only use service role key server-side ### 7. Don't Skip RLS Policies L' Disabling RLS for convenience �' Always use RLS, use service role key when needed ### 8. Don't Forget Cleanup L' Not unsubscribing from listeners �' Always cleanup in useEffect --- # 6. Template Version ## 6.1 Simplified Project Structure` ecomm-template/ �%%% app/ �% �%%% api/ �% �% �%%% checkout/route.ts �% �% �%%% webhook/route.ts �% �% �%%% send-order-email/route.ts �% �%%% auth/ �% �% �%%% page.tsx �% �% �%%% callback/route.ts �% �%%% checkout/ �% �% �%%% page.tsx �% �% �%%% success/page.tsx �% �%%% products/ �% �% �%%% [id]/page.tsx �% �%%% cart/page.tsx �% �%%% orders/page.tsx �% �%%% layout.tsx �% �%%% page.tsx �%%% components/ �% �%%% AuthProvider.tsx �% �%%% Navbar.tsx �% �%%% ProductCard.tsx �% �%%% ProductGrid.tsx �%%% lib/ �% �%%% supabase/ �% �% �%%% client.ts �% �% �%%% server.ts �% �%%% stripe.ts �% �%%% email/send.ts �%%% types/index.ts �%%% package.json `## 6.2 Minimal Implementation Checklist ### Core Features - [ ] Product listing - [ ] Product detail page - [ ] Add to cart - [ ] View cart - [ ] Checkout (Stripe) - [ ] Order creation (webhook) - [ ] Order history - [ ] Authentication (email/password + Google) - [ ] Order confirmation email ### Database Schema (Minimal)` sql -- Products CREATE TABLE products (id UUID PRIMARY KEY, name TEXT NOT NULL, price DECIMAL(10, 2) NOT NULL, image_url TEXT); -- Cart CREATE TABLE cart_items (id UUID PRIMARY KEY, user_id UUID REFERENCES auth.users(id), product_id UUID REFERENCES products(id), quantity INTEGER DEFAULT 1); -- Orders CREATE TABLE orders (id UUID PRIMARY KEY, user_id UUID REFERENCES auth.users(id), total DECIMAL(10, 2) NOT NULL, status TEXT DEFAULT 'processing', stripe_payment_intent_id TEXT UNIQUE); -- Order Items CREATE TABLE order_items (id UUID PRIMARY KEY, order_id UUID REFERENCES orders(id), product_id UUID REFERENCES products(id), quantity INTEGER, price DECIMAL(10, 2)); `### Environment Variables Required` env # Supabase NEXT_PUBLIC_SUPABASE_URL= NEXT_PUBLIC_SUPABASE_ANON_KEY= SUPABASE_SERVICE_ROLE_KEY= # Stripe NEXT_PUBLIC_STRIPE_PUBLISHABLE_KEY= STRIPE_SECRET_KEY= STRIPE_WEBHOOK_SECRET= # Resend RESEND_API_KEY= RESEND_FROM_EMAIL= # App NEXT_PUBLIC_APP_URL= ` ## 6.3 Quick Start Guide ### 1. Setup Supabase 1. Create Supabase project 2. Run SQL schema 3. Enable RLS 4. Create policies 5. Configure OAuth (Google) ### 2. Setup Stripe 1. Create Stripe account 2. Get API keys 3. Configure webhook endpoint 4. Get webhook secret ### 3. Setup Resend 1. Create Resend account 2. Verify domain 3. Get API key 4. Set FROM_EMAIL ### 4. Deploy to Vercel 1. Push to GitHub 2. Import to Vercel 3. Add environment variables 4. Deploy ### 5. Configure Domain 1. Add domain in Vercel 2. Update DNS records 3. Wait for propagation 4. Verify domain ## 6.4 Deployment Platforms ### Vercel (Recommended) - Pros: Best Next.js support, automatic deployments, free tier - Cons: Vendor lock-in - Best For: Next.js projects ### DigitalOcean App Platform - Pros: More control, competitive pricing - Cons: More setup required - Best For: Full control needed ### Railway - Pros: Simple, good DX - Cons: Newer platform - Best For: Quick deployments ### Render - Pros: Simple, good free tier - Cons: Slower cold starts - Best For: Budget-conscious projects ## 6.5 Template Customization ### Styling - Replace Tailwind with your preferred CSS - Update color scheme - Customize components ### Features - Add product categories - Add product reviews - Add user profiles - Add admin dashboard ### Integrations - Add analytics (Plausible, Google Analytics) - Add error tracking (Sentry) - Add monitoring (Vercel Analytics) --- # Conclusion This documentation covers the complete e-commerce project from architecture to implementation, problems to solutions, and concepts to code. Use this as a reference when building similar projects or learning Next.js, Supabase, Stripe, and modern web development. Key Takeaways: 1. Architecture Matters: Plan your stack and data flow 2. Security First: Use RLS, verify webhooks, protect secrets 3. Error Handling: Always handle errors gracefully 4. Testing: Test locally and in production 5. Documentation: Document as you build Next Steps: 1. Review the code in this project 2. Try building the template version 3. Customize for your needs 4. Deploy and iterate Happy Coding! =Ø€