SPOEQ Platform Architecture & Delivery Plan

1. High-Level Architecture Strategy

Start as a modular monolith (domain-separated folders + shared libs) to allow future extraction into microservices when scale/team demands. Separate domains: Auth, Catalog, Cart, Order, Payment, User Profile, Review, Inventory, Coupons, Analytics, Notification.

Layers

- Frontend Web (React + Tailwind + React Query)
- Admin Portal (React + MUI) – can be separate build or route segment
- API Gateway (Express) – routing, auth, rate limiting
- Domain Modules (service classes + controllers + repositories)
- Data Stores: MongoDB (primary), Redis (cache/session/pub-sub), Object Storage (S3/R2) for images
- Queue / Async: BullMQ (Redis) initially → RabbitMQ/Kafka later
- Observability: Winston logs → Loki, metrics (Prometheus) Phase 3
- Search (Phase 2+): Meilisearch / OpenSearch

Environments

Dev (hot reload, seeded data) / Staging (prod-like) / Prod (load balanced). CI builds Docker image per commit; staging auto-deploy; prod manual approval.

Scalability Path

1. Add Redis caching (product list, product detail, home aggregates)
2. Offload search to dedicated engine
3. Extract high-churn modules (Orders, Inventory) into services behind internal API or message bus
4. Shard MongoDB by userId (orders) & productId (reviews) if needed

2. System Architecture Diagram (Textual)

Browser & Admin -> API Gateway -> Domain Modules -> MongoDB / Redis / Queue / External APIs (Payments, Shipping, Email/SMS). Notifications & Analytics consume events from queue.

3. Core Flows

Checkout: Cart -> Auth check -> Address select -> Shipping method -> Payment intent -> Gateway redirect/confirm -> Webhook verify -> Mark paid -> Decrement inventory atomically -> Queue notifications -> Show confirmation.

Inventory Update: Order paid event -> Inventory module decrement -> If threshold crossed -> Low-stock notification -> Backorder prevention (stock >= requested in atomic query).

Review Submission: User verifies purchase (order item existence) -> Create review (pending) -> Moderator approves -> Product ratingAvg & ratingCount recomputed (aggregation or incremental update).

4. Frontend (Customer)

Pages: Landing, Category Listing, Product Detail, Cart, Checkout, Order Confirmation, Order Tracking, User Profile, Orders History, Wishlist, Reviews Management, Offers/Coupons, Help & Support, Auth (Login/Register/Forgot), Search Results.

State Management

- React Query for server cache (products, cart, orders)
- Lightweight global store (Zustand/Redux Toolkit) for UI ephemeral state (modals, toasts)
- URL drives filters (category?brand=..&price=..)

Components (Sample)

- ProductCard, ProductGallery, VariantSelector, PriceBlock, RatingStars
- FilterSidebar (facets), SortDropdown, PaginationControls
- CartItemRow, MiniCartDrawer, CouponApply
- AddressCard, PaymentMethodSelector, OrderSummary
- Timeline (order status), ReviewForm, WishlistGrid

Performance Practices

- Code splitting by route + prefetch next-route bundles
- Image lazy loading & responsive sizes
- Skeleton placeholders for product grid

5. Admin / Staff Portal

Modules: Dashboard, Product Management (CRUD, variants, bulk CSV import), Inventory (adjustments, logs, low-stock), Orders (search, status updates, refunds), Users (status/roles), Reviews Moderation, Coupons & Campaigns, Analytics (charts), Settings (tax/shipping/payment), Audit Log.

Roles & Permissions:

- Admin (all)

- Staff (no system settings or destructive account ops)

- Warehouse Manager (inventory + order fulfillment only)

- Support Agent (orders + user inquiries, limited refunds)

Enforcement: RBAC matrix (role -> allowed actions), middleware checks route metadata.

6. API Design (Representative)

Auth: POST /auth/register, /auth/login, /auth/refresh, /auth/logout, /auth/forgot-password, /auth/reset-password, /auth/oauth/:provider/callback

Products: GET /products (search/filter), GET /products/:id, POST/PATCH/DELETE (admin), GET /categories, GET /brands, POST /products/:id/images

Cart: GET /cart, POST /cart, PATCH /cart/:itemId, DELETE /cart/:itemId, POST /cart/apply-coupon

Wishlist: GET /wishlist, POST /wishlist, DELETE /wishlist/:productId

Orders: POST /orders, POST /orders/:id/pay, GET /orders/:id, GET /orders (user), PATCH /orders/:id/status (staff), POST /orders/:id/refund

Payments: POST /payments/intent, POST /payments/webhook, POST /payments/verify

Users: GET /me, PATCH /me, address CRUD

Reviews: POST /products/:id/reviews, PATCH /reviews/:id, DELETE /reviews/:id, GET /products/:id/reviews

Coupons: POST /coupons (admin), GET /coupons/:code (validate)

Analytics: GET /analytics/overview, /analytics/sales?range=, /analytics/top-products

Notifications: POST /notifications/subscribe (web push)

7. MongoDB Data Model

Users: { email, passwordHash, roles[], status, name, phone, addresses[], wishlist[], oauth{}, createdAt }

Products: { name, slug, categories[], brand, description, specs{}, images[], price{mrp,sale}, stock, sku, variants[], ratingAvg, ratingCount, tags[], createdAt }

Reviews: { productId, userId, rating, title, body, status, createdAt }

Cart: { userId, items[{ productId, variantKey, qty, priceSnapshot }], couponCode, updatedAt }

Orders: { userId, items[{ productId, variantKey, nameSnapshot, qty, unitPrice, subtotal }], pricing{}, status, payment{}, shippingAddressSnapshot{}, shipment{}, couponCode, createdAt }

Coupons: { code, type, value, minOrder, maxUses, perUserLimit, startAt, endAt, active, usedCount }

InventoryLogs: { productId, variantKey, delta, reason, refType, refId, createdAt }

AnalyticsDaily: { date, orders, revenue, newUsers, returningUsers, avgOrderValue }

Sessions/Refresh (optional separate collection if rotating tokens tracked)

Indexes

- users: email(unique), roles, status

- products: slug(unique), categories, brand, price.sale, ratingAvg, text(name, description)

- reviews: productId+rating, userId

- orders: userId+createdAt(desc), status, payment.intentId

- coupons: code(unique), active+startAt+endAt

- inventoryLogs: productId+createdAt

- analyticsDaily: date

Scaling & Consistency

- Stock decrement via findOneAndUpdate where stock >= qty

- Event to recompute product rating on review changes

- Caching: product detail & list composites in Redis with TTL & tag invalidation

8. Security & Payments

Auth: JWT access (15m), refresh (7d rotating), HTTP-only secure cookie for refresh, access token in memory.

Password: bcrypt (cost 12–14). Rate limit login attempts (Redis key per IP/email).

Validation: Zod/Joi schemas per route; centralized error translator.

Transport: HTTPS + HSTS; CSP & secure headers (helmet).

Payments: Verify Razorpay/Stripe signatures; store raw webhook payload + processed flag (idempotency). Order total recomputed on server (never trust client). Refund flow triggers payment gateway API then updates order/payment state.

Audit: Admin actions persisted with {actorId, action, resource, before, after, timestamp}.

Data Protection: Exclude passwordHash from projections, encrypt sensitive fields (optional field-level encryption for addresses/PII later).

9. Notifications & SEO

Notifications: Queue emails (order placed, shipped, delivered, refund, password reset), SMS for critical updates / COD verification, Web Push for shipment status & promos. Retry with exponential backoff; DLQ after N failures.

SEO: Pre-render landing, category, product pages (e.g. `@vitejs/plugin-ssr` or later Next.js migration). Structured Data (schema.org Product / Offer / BreadcrumbList). Sitemap generation daily. Canonical tags. OpenGraph meta. Image alt tags & compressed WebP/AVIF.

10. Logistics & Integrations

Shipping: Phase 1 manual tracking number entry. Phase 2 integrate Shiprocket (single provider). Phase 3 multi-carrier abstraction. Maintain shipment events array with {status, location?, timestamp, source}.

Address validation: Optional Phase 3 (India Post / Google Places API).

11. Future Enhancements (AI & Growth)

Recommendations: Collaborative filtering (purchase co-occurrence) -> Embedding-based similarity (vector DB) -> Contextual (in-session + user segment).

Search: Move to Meilisearch; synonyms (football=soccer), fuzzy matching. Later semantic rerank.

Chat Assistant: Retrieval over product specs + FAQ.

Loyalty: Points accrual per order, tiers (Bronze/Silver/Gold) with thresholds; redemption rules.

Marketplace: Add vendorId to product, vendor payout ledger, commission rates, vendor dashboard.

Mobile App: React Native reusing API & auth tokens.

Dynamic Pricing: Guardrail rules + rollback safety; A/B test.

12. Real-Time Features

Socket.IO namespaces: /orders (user-specific room order:<id>), /admin (dash metrics). Events: orderStatusUpdated, inventoryChanged. Redis Pub/Sub backend. Fallback via Server-Sent Events for tracking page.

13. Phased Execution Timeline

Phase 0 (Week 0–1): Repo, auth basics, product seed, CI/CD.

Phase 1 (2–4): Catalog, product detail, cart, checkout (COD + Razorpay), order create, minimal admin product CRUD.

Phase 2 (5–7): Addresses, reviews, coupons, inventory alerts, dashboard summary.

Phase 3 (8–10): Redis cache, websockets for order, analytics daily rollups, wishlist, rate limiting.

Phase 4 (11–13): Notifications pipeline (email/SMS/push), advanced filters, shipping API.

Phase 5 (14–16): Recommendations v1, search service extraction, performance tuning, A/B framework.

Phase 6+: Marketplace layer, mobile app, advanced AI and dynamic pricing.

14. Testing & Quality Strategy

Unit: utilities (pricing calc, coupon application, stock logic)

Integration: order placement end-to-end (mock gateway), review posting with rating update.

E2E: Playwright flows (browse→filter→detail→cart→checkout→track order).

Contract: API schema validation using OpenAPI + Dredd or Schemathesis.

Load: k6 scripts for /products, /checkout. Set baseline latency budgets.

CI Gates: lint, test, type check, vulnerability scan (npm audit / Snyk), build image.

15. Risks & Mitigations

Overselling stock → Atomic decrement & re-check before payment capture.

Webhook replay → Idempotent event table.

Slow catalog queries → Compound indexes + search engine + selective projections.

Fraud (COD spam) → COD order cap per user/day + SMS verification.

Data growth (reviews/orders) → Archival strategy or sharding.

16. Immediate Next Steps

1. Establish `api` folder structure (src/modules/{auth,users,products,orders}/...).

2. Install core deps (express, mongoose, zod, jsonwebtoken, bcrypt, redis, bullmq, winston, helmet, cors).

3. Implement User model + Auth routes (register/login/refresh) with tests.

4. Implement Product model + seed script + GET /products & GET /products/:id.

5. Implement Cart endpoints (server-synced) + frontend integration.

6. Integrate Razorpay sandbox: /payments/intent + webhook skeleton.

7. Basic admin product CRUD UI (protected route) in frontend.

8. Deploy staging (Docker compose: api, mongo, redis, frontend, nginx).

--- End of document.