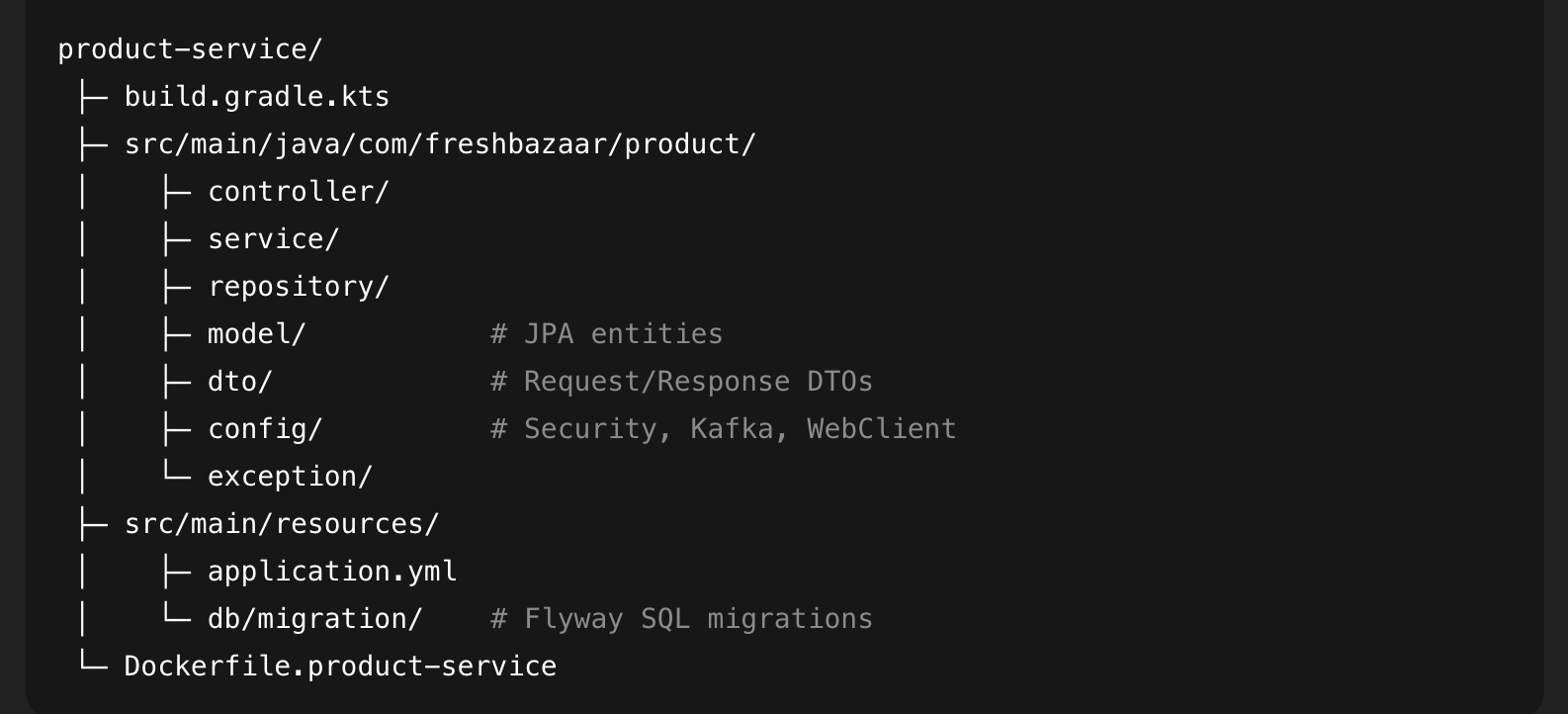
**Low-Level Design (LLD)**

**Low-Level Design** for **FreshBazaar** App covers the **backend** (service internals, classes, DTOs, persistence, events), **frontend** (components, API clients, state), **database** schemas + migrations, infra/devops bits (Dockerfile, CI pipeline sketch, Helm layout), and sequence/interaction flows for core use-cases.

**[ 1. ] Backend — per-service LLD**

Each service repo (module) uses same layout:

**Package naming:**

com.freshbazaar.product (or .auth, .trader, .customer).

## **Common infra classes (shared/common-utils)**

Put shared DTOs, error handling and utility code in shared/common-utils and publish as a local Maven/Gradle dependency to services:

ApiError { code, message, timestamp, details }  
BaseResponse<T> { success, data, error }

PagedResponse<T> { items, page, size, total }

**Auth-Service (LLD)**

**Entities (JPA):**

****

**RefreshToken entity for refresh tokens.**

### **DTOs:**

* RegisterRequest **{ email, password, role }**
* AuthResponse **{ accessToken, refreshToken, expiresIn, tokenType }**
* LoginRequest **{ email, password }**

### **Services:**

* **AuthService interface:**
  + AuthResponse **register(RegisterRequest)**
  + AuthResponse **login(LoginRequest)**
  + AuthResponse **refresh(String refreshToken)**
  + **boolean validate(String token)**

**Implementation notes:**

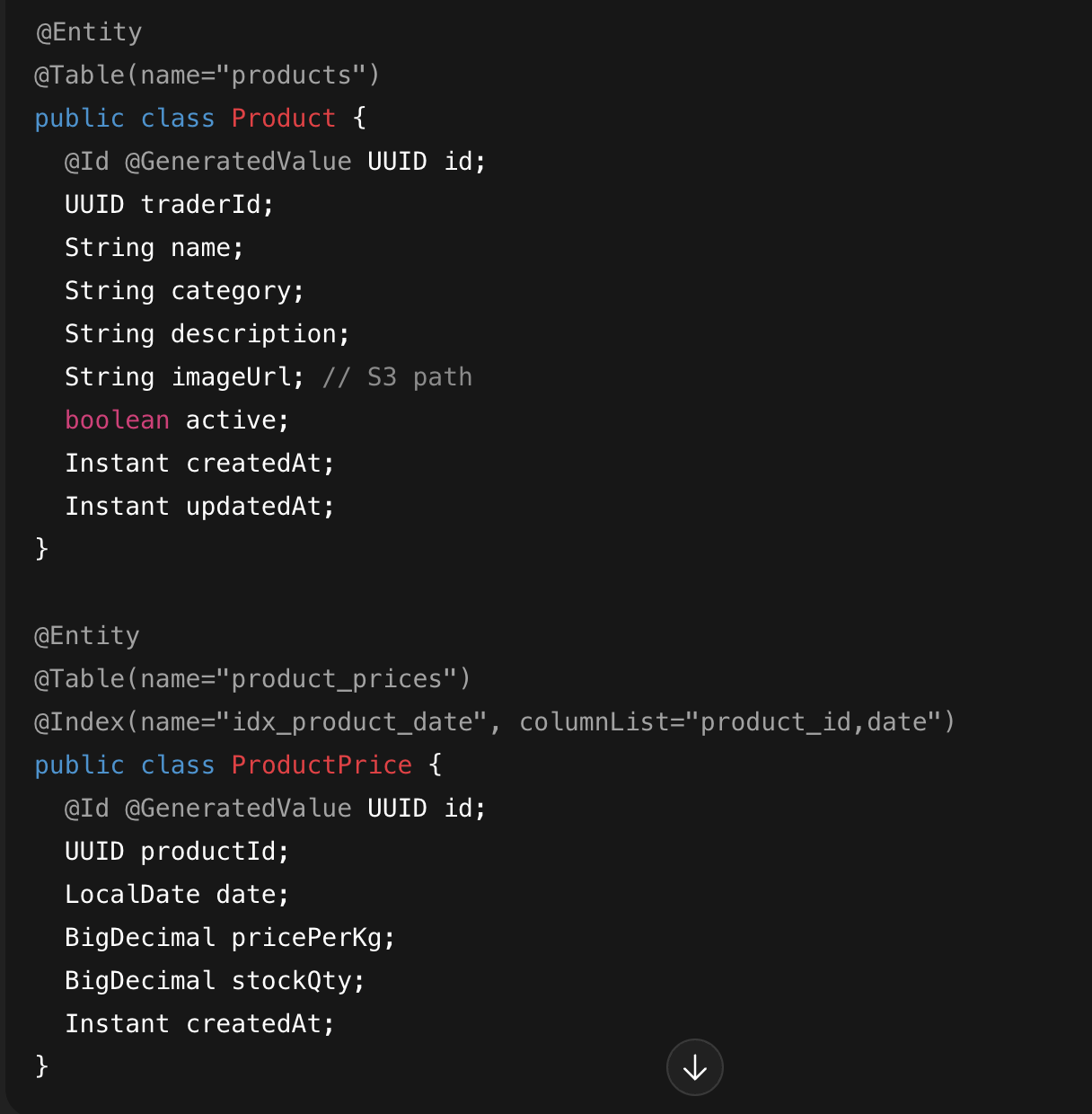
* Password hashing: BCrypt.
* JWT generation: include sub=userId, roles claim, exp. Sign with HMAC or RSA (prefer RSA for prod).
* Token revocation: store refresh tokens in DB; keep blacklist for invalidated access tokens (Redis with TTL) if required.

### **Security Config:**

* SecurityConfig registers:
  + JwtAuthenticationFilter — reads Authorization: Bearer ..., validates token, builds UsernamePasswordAuthenticationToken and sets security context.
  + /auth/\*\* permitted; other endpoints require authentication and role checks.
* Use MethodSecurity (@PreAuthorize("hasRole('TRADER')")) for service-level access control.

**Product-Service (LLD)**

**Entities:**

****

**Design note: product\_prices is append-only for audit + time-series; use partitioning by date or market\_id if scale grows.**

**DTOs (request/response):**

* CreateProductRequest **{ traderId, name, category, description,imageFile (multipart) }**
* ProductResponse **{ id, traderId, name, category, description, imageUrl, latestPrice, availability }**
* UpdatePriceRequest **{ productId, date, pricePerKg, stockQty }**
* PriceHistoryResponse **{ List<PricePoint { date, price } > }**

**Repositories:**

* **ProductRepository** extends **JpaRepository<Product, UUID>**
  + **custom: List<Product> findByCategoryAndActiveTrue(...).**
* **ProductPriceRepository** extends **JpaRepository<ProductPrice, UUID>**
  + **ProductPrice findTopByProductIdOrderByDateDesc(UUID productId) to get the latest price.**

**Services:**

**ProductService methods:**

* **ProductResponse createProduct(CreateProductRequest)**
  + **Validate trader exists (via Trader-service call or DB link).**
  + **Store image to S3 (async) → update imageUrl.**
  + **Save product entities.**
  + **Publish event product.created on Kafka with payload product id.**
* **void updatePrice(UpdatePriceRequest)**
  + Insert row into product\_prices.
  + Publish product.price.updated event (used to update ES index + notify clients).
* List<ProductResponse> searchProducts(SearchCriteria) — fetch from Elasticsearch or fallback to DB + aggregation.

**Controllers:**

**ProductController:**

* **POST /products (Trader) → multipart/form-data → create product.**
* **PUT /products/{id}/price → update price.**
* **GET /products → query params: category, minPrice, maxPrice, traderId, page, size.**
* **GET /products/{id} → product detail with latest price & history.**

**Eventing (Kafka):**

Topics:

* product-events with types: PRODUCT\_CREATED, PRICE\_UPDATED, PRODUCT\_UPDATED.

Consumers:

* search-service listens → update Elasticsearch index.
* notification-service listens → send push/websocket notifications to subscribers.
* analytics-service listens → update aggregations.

**Elasticsearch Mapping:**

Index products\_index fields:

* productId (keyword)
* traderId (keyword)
* name (text, analyzer)
* category (keyword)
* latestPrice (double)
* priceHistory (nested)
* location (geo\_point) if market-level filtering required

On PRICE\_UPDATED, produce ES document update:

* update latestPrice, append new priceHistory or maintain rolling window.

**Caching:**

* Cache frequent queries: top n products for market page in Redis (key market:top:marketId TTL 10s).
* Use @Cacheable on service methods, with invalidation when new price arrives.

**Error Handling:**

* Centralized @ControllerAdvice mapping exceptions to ApiError.
* Standard HTTP statuses:
  + 400 Bad Request
  + 401 Unauthorized
  + 403 Forbidden
  + 404 Not Found
  + 409 Conflict
  + 500 Internal Server Error

**Resilience:**

Use **Resilience4j** for:

* Circuit breaker on calls to other services (e.g., trader-service, search-service).
* Retry with exponential backoff for transient failures.
* Bulkhead patterns optionally.

## **Inter-service Contracts**

* Maintain OpenAPI YAML for each service in docs/api-specs/.
* Use contract tests (Pact or Spring Cloud Contract) in CI to ensure compatibility.

# **[ 2. ] Sequence flows (textual) — key user journeys**

**Trader posts product + price:**

1. Trader authenticates with POST /auth/login → receives JWT.
2. POST /products with image multipart + product JSON. Gateway routes to product-service (JWT validated).
3. **product-service**:
   1. stores image to S3 (synchronously or async with temporary URL),
   2. saves Product row,
   3. saves initial ProductPrice row,
   4. publishes product-events -> PRODUCT\_CREATED.
4. search-service consumes PRODUCT\_CREATED → indexes document in ES.
5. notification-service optionally sends WebSocket messages to subscribed clients.

**Customer views market rates:**

1. Customer app calls GET /products?marketId=...&sort=price\_asc.
2. Gateway forwards to product-service → product-service attempts to satisfy from Redis cache.
3. On cache miss, product-service queries Elasticsearch for filtered list, enriches with trader info via internal call to trader-service (or cached trader snapshot), returns paged data.
4. UI displays products with latestPrice and trader contact.

**Customer places a bid**

1. POST /bids (customer-service) with productId and bidPrice.
2. Save bid in bids table; publish bid.created.
3. Trader receives notification (email/SMS/push) through notification-service; trader can accept/reject using PUT /bids/{id}/status.

**[ 3. ] Database design (DDL + Flyway sample)**

**-– Flyway primary migration (V1\_\_initial\_schema.sql)  
—---------------------------------------------------------------------------------------------------------------------------  
CREATE TABLE users (**

**id UUID PRIMARY KEY,**

**email VARCHAR(254) UNIQUE NOT NULL,**

**password\_hash VARCHAR(255) NOT NULL,**

**enabled boolean default true,**

**roles text[],**

**created\_at timestamptz DEFAULT now()**

**);**

**CREATE TABLE traders (**

**id UUID PRIMARY KEY,**

**user\_id UUID REFERENCES users(id),**

**market\_id UUID,**

**name text,**

**address text,**

**contact\_phone text,**

**rating numeric(3,2) default 0,**

**created\_at timestamptz default now()**

**);**

**CREATE TABLE products (**

**id UUID PRIMARY KEY,**

**trader\_id UUID REFERENCES traders(id),**

**name text NOT NULL,**

**category text,**

**description text,**

**image\_url text,**

**active boolean default true,**

**created\_at timestamptz default now(),**

**updated\_at timestamptz**

**);**

**CREATE TABLE product\_prices (**

**id UUID PRIMARY KEY,**

**product\_id UUID REFERENCES products(id),**

**date date NOT NULL,**

**price\_per\_kg numeric(12,2) NOT NULL,**

**stock\_qty numeric(12,2),**

**created\_at timestamptz default now()**

**);**

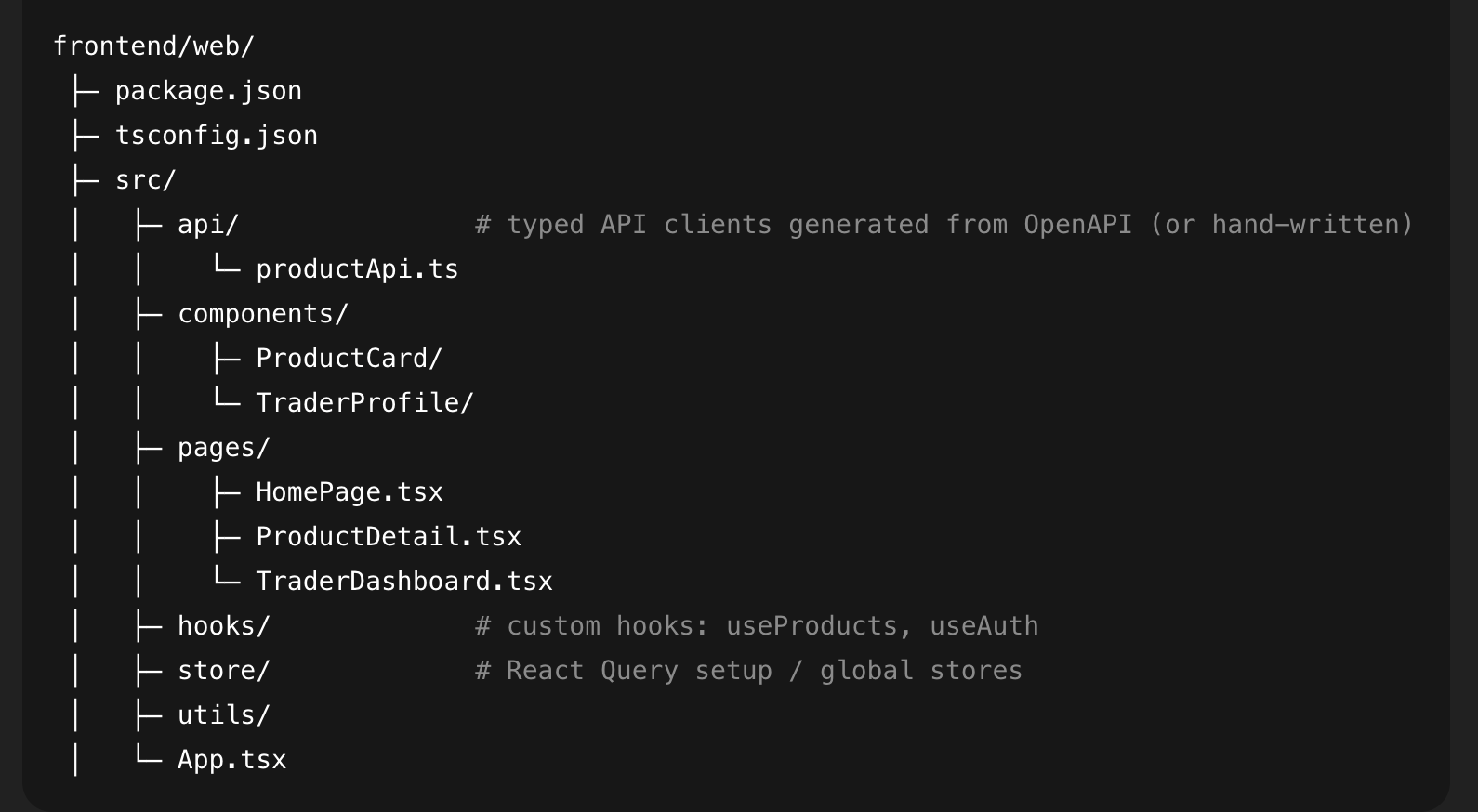
**CREATE INDEX idx\_product\_prices\_product\_date ON product\_prices(product\_id, date);**

**—---------------------------------------------------------------------------------------------------------------------------**

**Add migrations per feature; always write reversible or additive migrations.**

**[ 4. ] Frontend — low-level design**

**Project layout:**

****

## **API client:**

* Prefer generated TypeScript client from OpenAPI spec (keeps frontend & backend types aligned).
* Fallback: hand-coded axios client with typed interfaces.

**Example productApi.ts: **

## **State & caching:**

* Use **React Query** for server state (cache, background refetching, pagination).
* Use **localStorage** for auth tokens.
* Use **Context** for UI-only state (theme, modal state).

## **Real-time:**

* Use **WebSocket (STOMP)** to subscribe to /topic/market/{marketId} for live price updates.
* Reconcile WebSocket updates with React Query caches via queryClient.setQueryData(...).

## **Error handling & UI:**

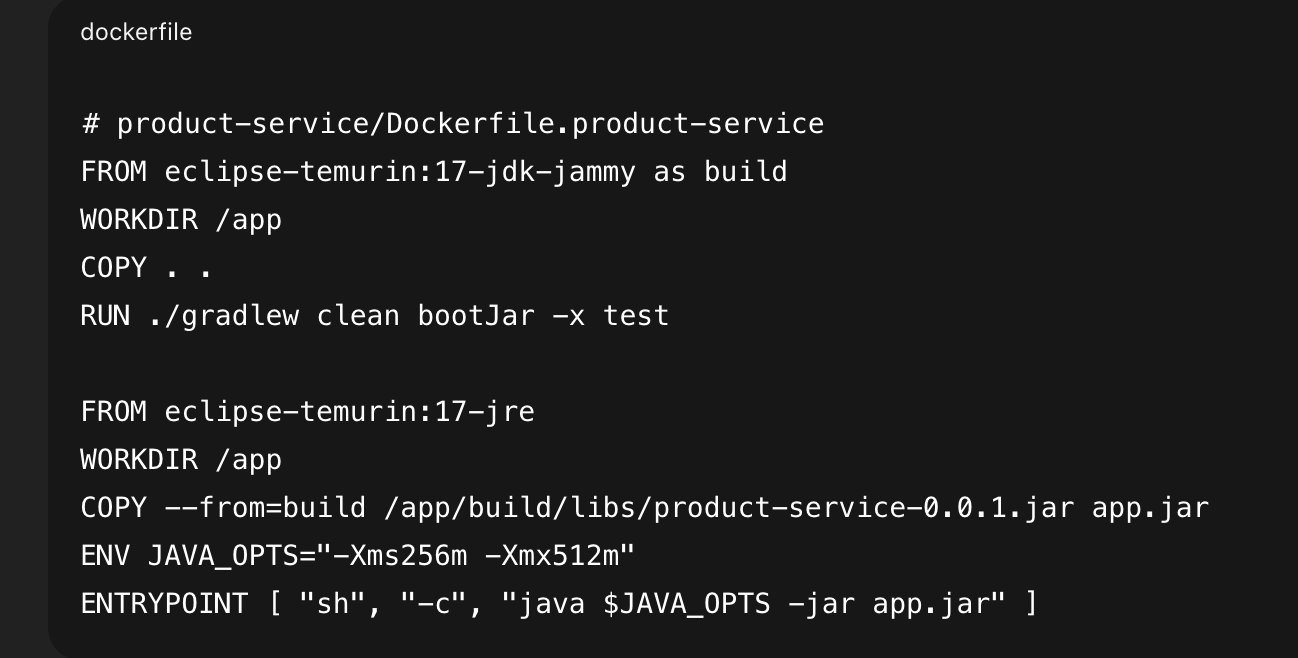
* Global ErrorBoundary for fatal errors.
* Toast notifications for success/error.
* Loading skeletons for product lists.

## **Tests:**

* Unit: Jest + React Testing Library
* E2E: Playwright with staging URL (run on CI for nightly pipeline)

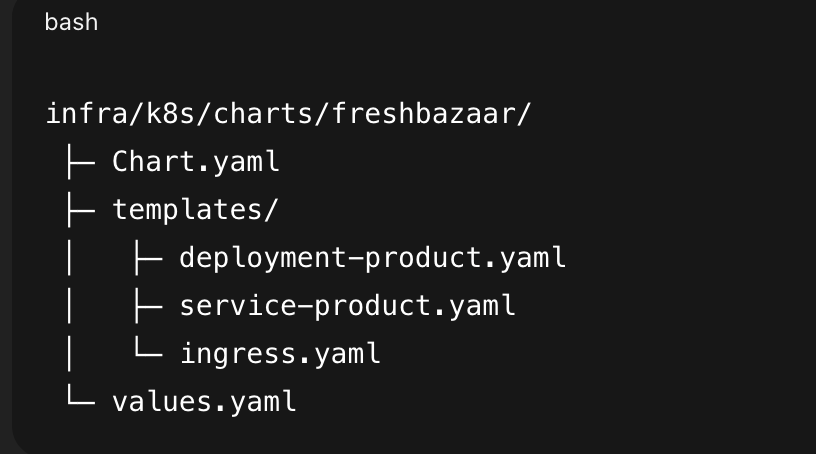
**[ 5. ] Infrastructure & CI/CD details**

**Dockerfile sample (product-service):**

****

**docker-compose (infra/docker/docker-compose.yml):**

* Services: postgres, redis, elasticsearch, kafka, zookeeper, minio, auth-service, product-service, gateway, frontend.
* Use environment variables for connection strings from the .env file.

**Kubernetes (Helm) skeleton:**

Use resource requests/limits and liveness/readiness probes.

**GitHub Actions (CI) — sketch:**

infra/ci-cd/github-actions/build.ymlruns on PR:

* Checkout code
* Set up JDK & Node
* For each service changed:
  + Run ./gradlew test (unit)
  + Run sonar-scanner (if applicable)
* For frontend: npm ci && npm test
* Build Docker images (on merge to main)
* Trigger security-scan.yml (sonar + veracode)

infra/ci-cd/github-actions/deploy.yml:

* Build & push images to registry
* Apply Helm chart to staging via kubectl (credentials from secrets)
* Run DB migrations (Flyway) as job or during startup

## **SonarQube & Veracode**

* Sonar: run per-project analysis using sonar-project.properties per service (or configure multi-module).
* Veracode: pipeline script infra/veracode/veracode-pipeline.sh invoked after build artifacts created.

## **Observability:**

* Instrument Spring Boot via **Micrometer** → Prometheus exporter.
* OpenTelemetry + Jaeger for traces.
* Forward logs to Splunk via Splunk Connect (k8s DaemonSet or sidecar).
* Define SLIs/SLOs in Grafana (latency, error rate).

# **Testing strategy:**

* Unit tests: every service uses JUnit 5 + Mockito.
* Integration tests: Spring Boot Test with Testcontainers for Postgres/Redis/Elasticsearch/Kafka.
* Contract tests: OpenAPI validation + consumer-driven contracts for critical inter-service flows.
* E2E: Playwright covering core flows (login, create product, view product list).
* Load tests: Gatling/k6 to test search and market-dashboard under load.

# **Operational concerns & best-practice patterns**

* Health endpoints: Spring Boot Actuator /actuator/health, /actuator/metrics.
* Secrets: never in repo; use Vault or cloud secret manager; CI pulls secrets from secure stores.
* Config: externalize via Spring Cloud Config or Kubernetes ConfigMaps + Secrets.
* Backups: PostgreSQL automated backups, S3 lifecycle policies.
* Rate limiting: API Gateway enforces per-user throttling.
* Feature flags: LaunchDarkly or Unleash for rolling releases of AI features.