**High-Level Design (HLD)**

**High-Level Design** for **FreshBazaar App** that covers **backend**, **frontend**, and **database**, and shows how to **integrate, scale, and deploy** the system.

**Backend High-Level Design**

1. **Core Architecture:**
   * Pattern: Microservices + API Gateway
   * Language/Framework: Java 21 + Spring Boot 3 (Gradle build)
   * Runtime: Docker containers, orchestrated via Kubernetes
   * Service Discovery & Config: Spring Cloud (Eureka optional) or Kubernetes service discovery; Spring Cloud Config / Vault for shared configs.
   * Communication: REST (JSON) + optional gRPC (internal heavy calls)
   * Security: Spring Security + JWT + OAuth2 (future: social logins)
2. **Microservices & Responsibilities:**

| **Service** | **Key Functions** | **DB-schemas** |
| --- | --- | --- |
| auth-service | User registration/login, token issuance & refresh, role mgmt (TRADER / CUSTOMER / ADMIN) | users, roles, refresh\_tokens |
| trader-service | Trader profiles, business details, rating avg cache | traders, addresses, licenses |
| product-service | CRUD for products, daily rate updates, image metadata, link to traders | products, product\_prices |
| customer-service | Customer profiles, favorites, contact requests, bids | customers, bids, favorites |
| review-service | Ratings & reviews for traders/products | reviews |
| search-service | Index to Elasticsearch for full-text and aggregated queries | ES index: products\_index |
| gateway-service | API Gateway + rate limiting, request auth, routing to internal services | N/A |

#### **Supporting Components -**

* Config Server: Centralized config.
* Message Bus (Kafka/RabbitMQ): Event-driven updates (e.g., “price changed” → search index update).
* Cache Layer (Redis): Fast retrieval of trending products, session store.

1. **Data Flow Example: Customer Viewing Market Rates:**
   * Frontend calls GET /api/products?marketId=X on Gateway.
   * Gateway verifies JWT → forwards to product-service.
   * product-service queries Redis cache; if miss, queries PostgreSQL or Elasticsearch for aggregated view.
   * Response returned to UI.
2. **Scaling & Fault Tolerance:**
   * **Stateless Services:** Horizontal scaling via Kubernetes HPA.
   * **DB:** PostgreSQL with read replicas; partitioning on market\_id if needed.
   * **Search:** Elasticsearch cluster with 3+ nodes.
   * **Images:** S3/MinIO with CloudFront CDN.
   * **Async jobs:** Kafka consumers for heavy tasks (price analytics, AI model inference)

**Frontend High-Level Design**

1. **Core Architecture:**
   * **Language**: TypeScript
   * **Framework**: React 18 + Vite
   * **State Management:** React Query (for API caching) + Context API
   * **UI Library**: Material UI (MUI) + Tailwind for utility styling
   * **Routing:** React Router v6
   * **Testing**: Jest + React Testing Library + Playwright (E2E)
   * **Build & Deploy:** Docker container served by Nginx or as a static artifact to CDN
2. **Modules / Pages:**

| Authentication | Login, Register, Password reset |
| --- | --- |
| Trader Dashboard | Add/Edit products, daily price update, upload images, analytics |
| Market Explorer | Product list, filters (category, price range), search, sorting |
| Product Details | Price history chart, contact trader, place bid |
| Reviews | Add/view trader or product reviews |
| Profile | Customer profile management, favorites, settings |
| Admin (future) | Manage traders/customers, moderate reviews |

1. **Integration Approach:**
   * **API Layer**: Axios + typed endpoints (/src/api) with TypeScript interfaces matching backend OpenAPI specs.
   * **Real-time Updates:** WebSockets (via Spring Boot STOMP) or SSE for live price updates.
   * **PWA Ready:** Service Worker for offline caching and “Add to Home Screen” on mobile.
2. **Scalability & Delivery:**
   * Static build served from CDN (CloudFront) for low latency.
   * Separate mobile app later: React Native reusing TypeScript types & API clients.

**Database & Data Design**

1. **Primary Database: PostgreSQL:**
   * **Reason:** ACID transactions, JSONB support for semi-structured data, mature tooling, easy horizontal read scaling.
   * **Schema Highlights**

* users (id, email, password\_hash, role, created\_at)
* traders (id, user\_id FK, market\_id, name, address, contact\_info, rating\_cache)
* products (id, trader\_id FK, name, category, description, image\_url)
* product\_prices (id, product\_id FK, date, price\_per\_kg, stock\_qty)
* bids (id, customer\_id FK, product\_id FK, bid\_price, status)
* reviews (id, reviewer\_id FK, trader\_id FK, rating, comment)
  + **Scaling Plan:**
* Start with one write node + read replica.
* Partition product\_prices by market\_id and date for large datasets.
* Use TimescaleDB extension if time-series analytics become heavy.

1. **Search & Analytics:**
   * **Elasticsearch**: For full-text search, aggregations (min/max/avg prices).
   * **Redis:** Cache frequently accessed products, user sessions, and trending queries.
2. **Object Storage:**
   * **AWS S3 (or MinIO in dev)**: Stores product images and documents with lifecycle rules for cost control.

**Integration Plan**

| **Layer** | **Integration Method** | **Notes** |
| --- | --- | --- |
| Backend ↔ Frontend | REST + WebSocket | API contracts documented via OpenAPI; CI enforces schema compatibility. |
| Services ↔ Services | REST + Kafka events | Event-driven updates (e.g., price change triggers search index refresh). |
| CI/CD ↔ Quality Tools | GitHub Actions → SonarQube, Veracode | Quality and security gates on every merge. |
| Logs & Metrics | OpenTelemetry → Splunk/Prometheus/Grafana | Unified monitoring, distributed tracing. |

**Continuous Integration:**

* GitHub Actions: build, test, Sonar scan, Docker image push.
* Feature branches → Pull Request → automatic quality gate.

**Continuous Deployment:**

* ArgoCD or Flux for GitOps to Kubernetes.
* Canary or blue/green deployments for zero-downtime releases.

**Deployment & Scaling Strategy**

1. **Environments:**
   * **Dev**: Docker Compose (local)
   * **Staging**: Single-node Kubernetes (or lightweight cluster)
   * **Prod**: Managed Kubernetes (AWS EKS / GKE / AKS)
2. **CI/CD Pipeline:**
   * Build & Unit Test
   * SonarQube Analysis
   * Veracode/Snyk Security Scan
   * Docker image build & push
   * Deploy to Staging
   * Manual approval → Production
3. **Observability:** 
   * Logs: Splunk
   * Metrics: Prometheus + Grafana
   * Alerts: PagerDuty/Slack integration
   * Tracing: OpenTelemetry + Jaeger
4. **Scaling:** 
   * **Stateless Microservices:** Horizontal Pod Autoscaler based on CPU/RPS.
   * **DB Scaling:** PostgreSQL read replicas, partitioning; Elasticsearch cluster sharding.
   * **CDN:** CloudFront caches frontend assets and S3 images globally.

## **Key Takeaways**

* Backend is stateless microservices with Spring Boot + Gradle, scaling horizontally in Kubernetes.
* Frontend is TypeScript React served via CDN, designed for easy future extension to React Native mobile apps.
* Database uses PostgreSQL for core transactional data, Elasticsearch for search/analytics, Redis for caching.
* Integration is event-driven (Kafka) plus REST, with strict OpenAPI contracts and CI checks.
* Deployment is CI/CD → Docker → Kubernetes with SonarQube, Veracode, and Splunk integrated, enabling safe automated scaling and real production reliability.

# **Conventions & tech choices:**

* Java 17, Spring Boot 3, Gradle (Kotlin DSL) for services.
* TypeScript + React 18 + Vite for frontend.
* PostgreSQL (primary), Elasticsearch (search), Redis (cache), S3/MinIO (images).
* Kafka for events.
* JWT auth (Spring Security).
* Flyway for DB migrations.
* Docker + Kubernetes (Helm) for deployment.
* SonarQube + Veracode integrated in CI, Splunk for logs, Prometheus/Grafana for metrics.