#### ****1. High-Level Architecture Diagram****

The architecture will include the following components:

* **Frontend UI**: User interface for store managers/employees (React.js or Angular).
* **Backend Services**:
  + **Inventory Service**: Manages inventory data, reservations, and alerts.
  + **Sync Service**: Handles data consistency and synchronization between stores.
  + **Transfer Service**: Handles stock transfers.
* **Database**: Relational (e.g., PostgreSQL) and a distributed cache (e.g., Redis) for real-time updates.
* **Message Broker**: Apache Kafka or RabbitMQ for inter-service communication.
* **Monitoring Tools**: Prometheus & Grafana for monitoring and alerting.

#### ****2. Class Diagram****

**Entities & Relationships**:

**Store**:

* store\_id (Primary Key)
* store\_name
* location

**Product**:

* product\_id (Primary Key)
* product\_name
* capacity

**Inventory** (Mapping store and product):

* store\_id (Foreign Key to Store)
* product\_id (Foreign Key to Product)
* current\_stock

**Reservation**:

* reservation\_id (Primary Key)
* product\_id (Foreign Key to Product)
* store\_id (Foreign Key to Store)
* reserved\_quantity
* reservation\_time

#### ****3. API Specifications for Core Operations****

**Track Inventory Levels**

**Endpoint**: GET /inventory/{storeId}

**Response**:

json

{

"storeId": "S001",

"products": [

{ "productId": "P001", "name": "Product A", "stock": 100, "capacity": 200 }

]}

**Create Product Reservation**

**Endpoint**: POST /reservation

**Request Body**:

json

{

"productId": "P001",

"storeId": "S001",

"quantity": 5 }

**Response**: 200 OK or 400 Bad Request with reason.

**Inventory Transfer**

**Endpoint**: POST /transfer

**Request Body**:

json

{

"fromStoreId": "S001",

"toStoreId": "S002",

"productId": "P001",

"quantity": 10 }

* + **Response**: 200 OK or 404 Not Found.

**Stock Alerts**

* + **Endpoint**: GET /alerts
  + **Response**: Alerts if stock is below threshold (20%).

#### ****4. Technology Choices & Trade-Offs****

**Frontend**

* **React.js**: For a dynamic, user-friendly interface.
  + Widely adopted, good ecosystem, and developer productivity.

**Backend**

* **Spring Boot (Java)**: Robust microservices with REST APIs.
  + Provides scalability, stability, and developer tools.

**Database**

* **PostgreSQL**: For relational data consistency.
  + Handles complex transactions and supports indexing for faster lookups.
* **Redis**: For caching real-time inventory queries and reservations.

**Messaging**

* **Apache Kafka**: Distributed event streaming for inventory synchronization.

**Monitoring**

* **Prometheus** & **Grafana**: For real-time monitoring and visualization.

#### ****5. Handling Concurrency and Real-Time Updates****

* **Optimistic Concurrency**: Versioning in database records to handle concurrent updates.
* **Distributed Locking**: Redis-based locking for reservations and stock transfers.
* **Event-Driven Updates**: Inventory updates via Kafka topics to sync store data in real time.

#### ****6. Edge Case Handling****

* **Over-Reservations**: Lock the quantity before confirmation. Expire holds after 30 minutes using TTL in Redis.
* **Network Failures**: Retry logic for API calls and use circuit breakers (e.g., Hystrix).
* **Partial Transfers**: Rollback operations if one part of a multi-step process fails.