PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

COURSE: SOFTWARE ARCHITECTURE
Final Report

INSTRUCTOR:	Vũ Quang Dũng
GROUP 6:	Nguyễn Duy Khánh(23010335 – K17-KTPM)
	Nguyễn Thị Lan Anh (23010283 – K17-CNTT8)
	Trần Trọng Minh (23010563 – K17-CNTT)
	Nguyễn Hải Đăng (23010099 – K17-)
CREDIT CLASS:	CSE703110-1-2-25(N02)

Ha Noi, January 6, 2025

Preface

In the context of rapid digital transformation, the application of information technology in business management—specifically within the F&B (Food and Beverage) sector—is no longer just an option but an inevitable trend. Traditional management systems, or legacy Monolithic systems, often face significant challenges regarding scalability and real-time data synchronization as the volume of customers and transaction data continues to grow.

Recognizing these challenges, our team decided to undertake the project titled **"Starbuzz ERP: A Microservices-based Management System"**. This project focuses on addressing the management of sales, inventory, and staff coordination by decomposing the system into independent services that communicate flexibly via standard RESTful APIs and WebSockets.

A primary driver for this project is the strategic goal to **minimize leftover food** and waste through intelligent inventory tracking. This report details the system analysis, design, and implementation process, utilizing a modern technical stack including **Python**, **React**, and **Flutter**. We hope that the results of this project will serve as a useful reference for applying modern software architecture to real-world business problems.

We hope that the results of this project will serve as a useful reference model for the application of modern software architecture to real-world management problems.

To complete this project, alongside the efforts of our team members, we have received dedicated attention, encouragement, and guidance from Vu Quang Dung. Throughout the implementation of this topic, he spared no effort in imparting knowledge, guiding our problem-solving mindset, and providing valuable feedback to help us refine the system, ranging from Microservices organization to source code optimization.

Although the team has made every effort to apply learned knowledge to practice, due to time constraints and limited practical experience, the project inevitably contains shortcomings. We sincerely look forward to receiving your feedback to further improve the topic and gain valuable lessons for our future careers.

We would like to express our sincere gratitude!

I.Introduce

1. Cover pages & Info

-Project Title:Starbuzz ERP – Coffee Shop Management System.
-Course Name: Software Architecture
-Student Info:Nguyễn Duy Khánh(23010335 – K17-KTPM)
Nguyễn Thị Lan Anh (23010283 – K17)
Trần Trọng Minh (23010563–K17)
Nguyễn Hải Đăng (23010099 – K17)
-Date:February,2026

1.2.Project scope

-Product name: Starbuzz ERP.
-Users: Store managers, service staff, bartenders, and warehouse personnel.
-Main goal: Automate the sales process with a special focus on minimizing food waste through smart inventory management.

1.3.Documentation Conventions

-ERP: Enterprise Resource Planning.
-API: Application Programming Interface.

2. Executive Summary

The project focuses on the design and implementation of **"Starbuzz ERP,"** a comprehensive Coffee Shop Management System aimed at optimizing operational workflows and resource management. The system addresses core requirements for managing product menus, staff coordination, and real-time order processing. A strategic priority of this project is the integration of intelligent inventory tracking to achieve the mission of **minimizing leftover food** and reducing waste in the F&B environment.

Throughout the development process, the system has successfully transitioned from a traditional Monolithic architecture to a modern **Microservices Architecture** to ensure high scalability and easier maintenance.

The final system comprises:

+Independent Microservices: Identity, Order, Product, Inventory, and Reporting Service.

+Modern Frontend & Mobile App: Built using a **Single Page Application (SPA)** for administrative tasks and a **Flutter-based mobile application** for staff, enabling smooth, real-time interactions without page reloads.

+Centralized Database with Logical Separation: The system utilizes a shared MySQL database schema where each microservice—such as the Inventory or Order service—accesses only the tables relevant to its specific business domain. This design ensures clear data ownership while remaining transition-ready for a full Database-per-Service model.

+Hybrid Communication: Combines **REST API** for synchronous requests and **WebSockets (Socket.IO)** for asynchronous, real-time broadcasting of orders between the point-of-sale and the barista station.

The achieved result is a system with high availability and low latency, capable of handling peak traffic volumes during busy hours while providing precise data to support sustainable management goals.

2.1. Architectural Overview

This section outlines the high-level architecture of the **Starbuzz ERP** system. We have adopted a **Monorepo** structure combined with **Docker** containerization to ensure a unified development lifecycle and consistent deployment environments.

2.1.1. Architectural Pattern

The system follows a classic **Three-Tier Architecture**, modernized with containerization:

+**Presentation Layer (Frontend):** A Single Page Application (SPA) built with React.js.

+**Application Layer (Backend):** A RESTful API built with Node.js and Express.js.

+**Data Layer (Database):** A document-oriented database using MongoDB.

The application components interact as follows:

1. **User Interaction:** Users interact with the React Client via HTTP requests.
2. **API Communication:** The Client sends JSON requests to the Express Server.
3. **Data Processing:** The Server processes logic and uses Mongoose to query the Database.
4. **Storage:** MongoDB handles persistent data storage and retrieval.

2.2. Technology Stack Selection

The following technologies were selected to meet the requirements of real-time performance and scalability.

Component	**Technology**	**Description**
Frontend	**React.js**	chosen for its component-based architecture and efficient DOM updates.

Backend	**Node.js + Express**	chosen for its non-blocking I/O model, suitable for handling concurrent POS transactions.
Database	**MongoDB**	chosen for its flexible schema (NoSQL), allowing easy adaptation to changing product attributes.
ODM	**Mongoose**	used to provide schema validation and organize data logic within the application.
DevOps	**Docker Compose**	used to orchestrate the multi-container environment (Client, Server, DB).

2.3. Backend Design (Application Layer)

The backend logic is centralized within the server directory and exposes data via RESTful endpoints.

2.3.1. Project Structure

The backend structure isolates data models from business logic (routes).

Plaintext

```
/server
  ├── models/       # Mongoose Schemas (e.g., Staff.js, Order.js)
  ├── routes/       # API Routes (api.js)
  ├── server.js     # Entry point & DB Connection
  └── Dockerfile    # Backend container config
```

2.3.2. Key API Specifications

The system exposes the following primary API groups:

+**POS Operations:** Endpoints to fetch products (GET /products) and submit orders (POST /orders).

+**Human Resources:** Endpoints to manage staff profiles (CRUD /staff) and track attendance (POST /attendance).

+**Inventory:** Endpoints to monitor and update stock levels (GET/POST /inventory).

2.4. Database Design (Data Layer)

The database schema is designed using **Mongoose** to ensure data integrity within the NoSQL environment.

2.4.1. Core Data Models

The system relies on the following key entities:

+**Staff:** Stores employee details, roles (Manager/Staff), and shifts.

+**Attendance:** Linked to Staff via ObjectId, tracking check-in/out times.

+**Product:** Stores menu items, prices, and categories.

+**Order:** Stores transaction details, including lists of items and total revenue.

+**Ingredient:** Manages inventory stock levels and minimum thresholds.

2.4.2. Schema Example (Staff)

JavaScript

```
const StaffSchema = new mongoose.Schema({
  code: { type: String, required: true, unique: true },
  name: { type: String, required: true },
  role: String,
  shift: String,
  salary: Number,
  status: { type: String, default: 'offline' }
```

});

2.5. Frontend Design (Presentation Layer)

The frontend is designed to serve two distinct user groups: Administrators and POS Operators.

2.5.1. Layout Strategy

+**Admin Dashboard (AdminLayout):** Features a persistent sidebar for navigation between management modules (Inventory, Staff, Reports).

+**POS Interface (POSPage):** A full-screen, grid-based layout optimized for speed. It splits the screen into a **Product Grid** (Left) and a **Transaction Cart** (Right).

2.5.2. State Management

+**Global State:** We utilize React **Context API** (CartContext) to manage the shopping cart state and persist it across component re-renders.

+**Data Fetching: Axios** is used to communicate with the Backend API.

2.6. Infrastructure & Deployment

The system infrastructure is defined as code (IaC) using Docker.

2.6.1. Docker Configuration

The docker-compose.yml file orchestrates three services:

1. **mongo:** Persists data using a named volume (mongo-data).
2. **server:** Depends on mongo; exposes API on port 5000.
3. **client:** Depends on server; serves the UI on port 3000.

3. Project Requirements & Goals

3.1 Core Functional Requirements

ID	Function	Detailed Description
FR-01	Inventory & Menu Management	Administrators can perform CRUD operations (Create, Read, Update, Delete) on product information and raw material catalogs.
FR-02	Real-Time Ordering (POS)	Staff can log in to the Flutter mobile application to create and send orders. The system validates stock availability before confirming the transaction.
FR-03	Automated Waste Management	The system tracks ingredient expiration dates and usage patterns. It automatically triggers alerts for items nearing expiry to **minimize leftover food**.
FR-04	Asynchronous Status Updates	The system utilizes WebSockets to send instant notifications to the Barista station once a new order is placed (background processing).
FR-05	Business Analytics & Reporting	Managers can view detailed reports on daily revenue, ingredient consumption rates, and a "Waste Log" to track discarded items.
FR-06	Security & Identity Management	The system enforces strictly controlled access (Manager/Staff/Barista) through a dedicated Identity Service using secure authentication.

3.2 Key Quality Attributes

Performance & Scalability

Particularly critical for UC1 (Real-Time Ordering) and peak hours in the coffee shop.

> **+NFR-01 (Response Time):** The system must respond to data read requests (GET) within < 2 seconds. For write transactions (e.g., placing an order, updating stock), processing time must not exceed 2 seconds.
>
> **+NFR-02 (Concurrency / Load Capacity):** The system must support a minimum of 100 - 500 Concurrent Users (CCU) during peak morning hours without crashing or service interruption.
>
> **+NFR-03 (API Gateway Latency):** The API Gateway must not introduce a latency exceeding 50ms when routing requests from the Client to backend Microservices (Order, Inventory, Identity).
>
> **+NFR-04 (Asynchronous Processing):** Time-consuming tasks, such as generating end-of-day waste reports or importing large supplier lists, must be handled as Background Tasks (Asynchronous) to prevent blocking the user interface.

Security

Crucial as the system stores staff personal information and business revenue records.

> **+NFR-05 (Authentication & Authorization):** All APIs accessing resources must be protected via JWT (JSON Web Token) mechanisms. The API Gateway must reject requests without a valid token.

+NFR-06 (Role-Based Access Control - RBAC): The system must strictly enforce authorization based on roles:

-Staff: Can only view the menu and create orders for customers.

-Barista: Can only view and update the status of drink preparation.

-Manager/Admin: Has full administrative privileges, including pricing and revenue reporting.

+NFR-07 (Data in Transit Security): All communication between the Client (Web/Mobile) and Server (API Gateway) must be encrypted via HTTPS protocol (TLS 1.2 or higher).

+NFR-08 (Password Hashing): User passwords in the Identity Service must be hashed using strong algorithms such as BCrypt or Argon2.

Reliability & Availability

Ensures stable shop operation and consistent inventory data.

+NFR-09 (Availability): The system guarantees an Uptime of 99.5% during business hours to ensure sales are never interrupted.

+NFR-10 (Fault Tolerance): Failure in non-critical services (e.g., Email notification for low stock) must not interrupt core order processing.

+NFR-11 (Data Consistency): The system ensures strong consistency in the Inventory workflow. Slot availability and raw material stock must be updated within atomic transactions to avoid "out-of-stock" errors during peak ordering.

Maintainability & Extensibility

Serving the Development Team using modern standards.

+NFR-12 (Microservices Architecture): Services (Order, Inventory,

Product, etc.) must be independently deployable. Updating the code of the Inventory Service must not require restarting the Order Service.

+NFR-13 (API Standard): APIs must adhere to RESTful standards, correctly utilizing HTTP Methods and Status Codes (200, 201, 400, 404, 500).

+NFR-14 (Code Quality): Python source code must comply with PEP 8 standards, with full comments for complex logic such as automated waste calculation and recipe-to-stock deductions.

+NFR-15 (Logging): The system must have a centralized Error Logging mechanism to facilitate rapid debugging during store operations.

Usability

Serving the End-User Experience for Staff and Baristas.

+NFR-16 (Responsive Design): The interface must display correctly on both Desktop (Management) and Mobile/Tablet (Staff POS), especially for order entry and stock checking.

+NFR-17 (User-Friendly Error Messages): If a system error occurs, the interface must display an understandable message (e.g., "Payment system busy, please try again") instead of technical stack traces.

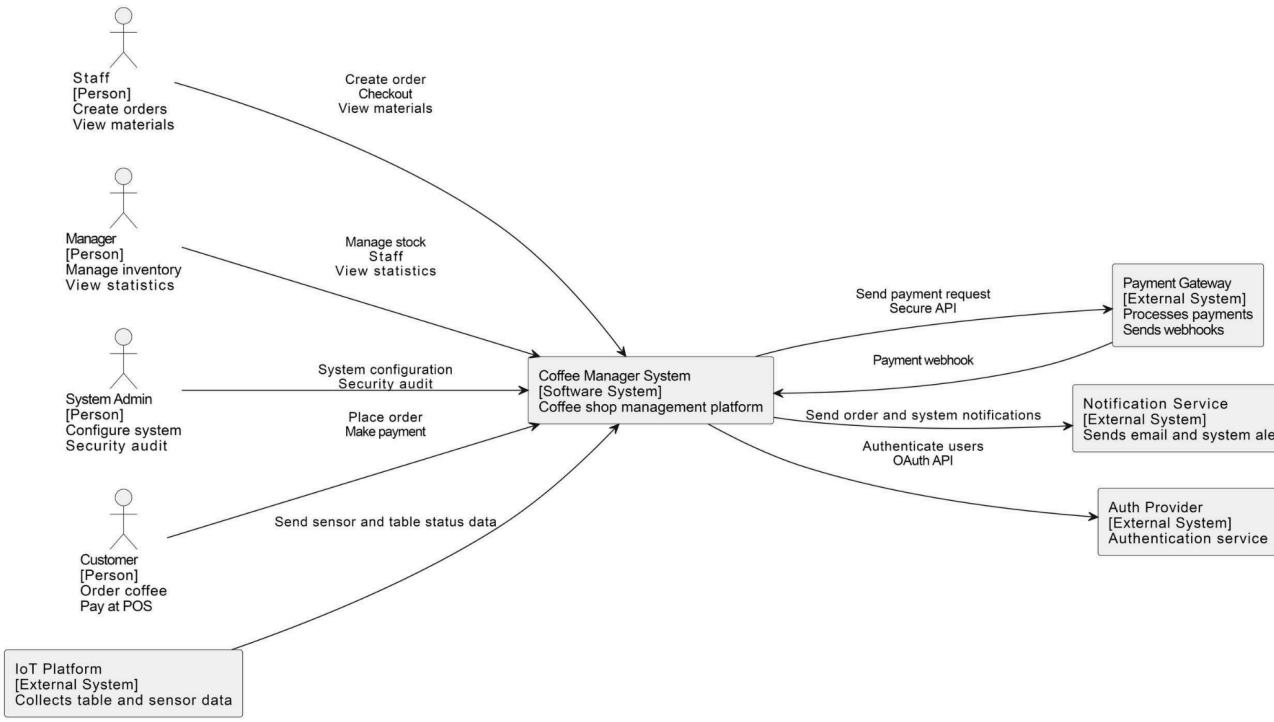
+NFR-18 (Minimal Interaction/Efficiency): Staff can process a "Quick Pay" for frequent orders with a single interaction to speed up the checkout process during busy periods.

3. Architectural Design

In this project, the **C4 Model** (Context, Container, and Component) and UC are utilized to provide a clear, hierarchical view of the **Starbuzz ERP** architecture, moving from a high-level overview to detailed service implementation.

3.1. System Context Diagram (C4 Level 1)

"The **Starbuzz ERP** acts as the central nervous system for coffee shop operations. The **Staff/Barista** interacts with the system to capture customer orders and modify item availability in real-time. The **Manager** utilizes the system to access high-level analytics, such as peak-hour sales and inventory depletion rates. Externally, the system integrates with the **Banking Payment Gateway** via a secure RESTful API to facilitate cashless transactions. It also communicates with an **IoT Sensor Hub** (if applicable) to monitor table occupancy and environmental conditions of the shop."



Overview: This diagram illustrates the high-level interactions between the **Coffee Management System (Starbuzz ERP)** and its external environment.

Actors:

+Staff/Manager: Interacts with the system to process orders, manage stock, and view real-time business reports.

+Customer: Interacts with the system through the POS terminal or by

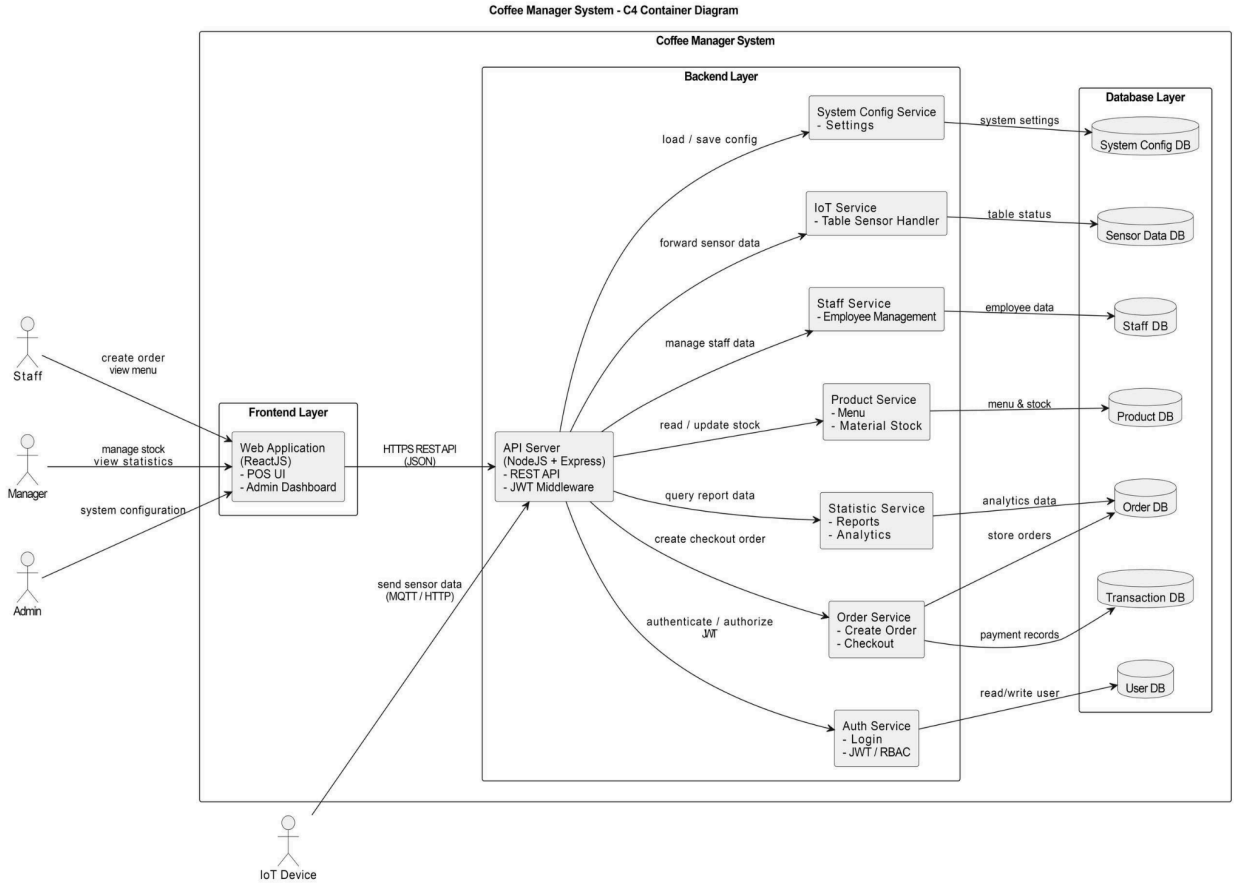
scanning QR codes for payment.

+**System Admin:** Manages system configurations, user roles, and security audits.

-**External Systems:**Payment Gateway: A third-party service (e.g., Bank API) that processes QR code transactions.

Auth Provider: Handles secure user authentication (via JWT or OAuth).

3.2. Container Diagram (C4 Level 2)



"The architecture is decomposed into several functional containers to ensure that a failure in one service (e.g., the Statistic Service) does not halt the core ordering process.

-Key Containers:

+**Web Application (ReactJS):** A single-page application (SPA) that provides a low-latency interface for POS operations.

+**API Gateway (Node.js/Express):** Serves as the traffic cop. It handles cross-cutting concerns such as **JWT validation** and **Request Rate Limiting** to prevent system abuse.

+**Order Service:** Orchestrates the lifecycle of a coffee order from 'Draft' to 'Completed'. It manages the business rules for discounts and VAT calculations.

+**Inventory Service:** Tracks real-time stock levels. When an order is placed, it automatically decrements the raw materials (e.g., coffee beans, milk).

+**Database Layer:** Each service is backed by its own **MySQL/PostgreSQL instance**, ensuring data isolation and allowing for independent schema migrations."

3.3. Component Diagram (C4 Level 3)

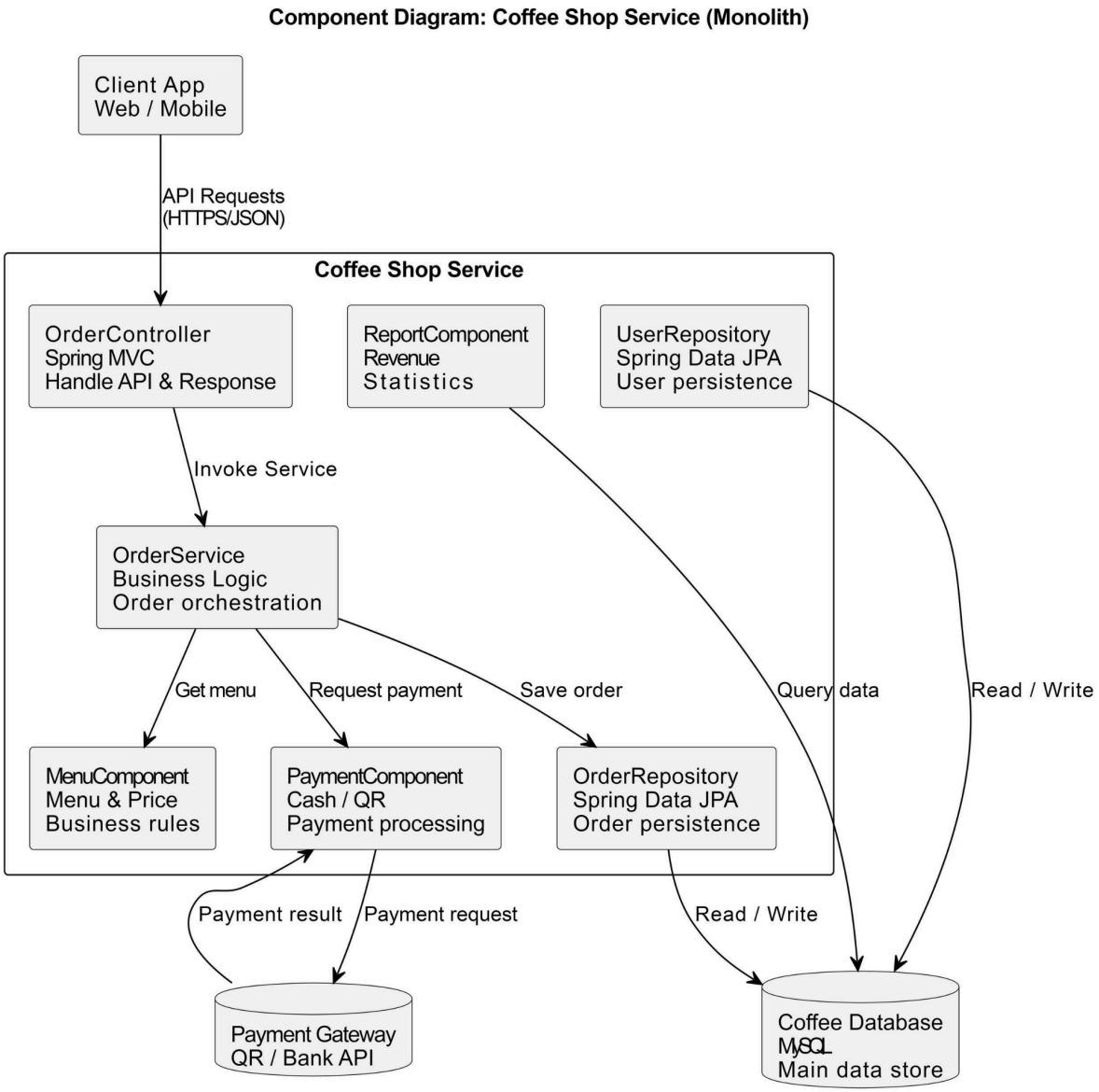
"Inside the **Order Service**, the logic is divided into modular components:

+**OrderController:** Defines the REST endpoints (e.g., `POST /api/orders`) and performs initial DTO (Data Transfer Object) validation.

+**PricingEngine:** A specialized component that calculates the total price by applying active promotions and membership level discounts.

+**PaymentHandler:** Manages the state machine for payments. It coordinates with the external Payment Service to verify successful QR code scans before marking an order as 'Paid'.

+**Repository Layer:** Abstracted using **Sequelize/TypeORM**, this layer handles the complex SQL joins required to fetch order history while protecting the system from SQL injection."



3.4.Software Design & Code Implementation(C4 level 4)

3.4.1. Controller Layer: API Endpoint Management

The Controller layer acts as the entry point for all external communications, ensuring that requests are properly validated before reaching the core business logic.

+**OrderController**: Responsible for handling order-related actions such as `POST /orders` and `POST /orders/checkout`.

+**PaymentController**: Manages financial interactions, specifically for generating QR codes (`/payments/qr`) and receiving status updates via banking webhooks (`/payments/webhook`).

3.4.2. Service Layer: Business Logic & Orchestration

The Service layer contains the "intelligence" of the system, coordinating data between different domain services and the database.

+**OrderService**: Manages the end-to-end lifecycle of a customer's order. It orchestrates the flow from initial creation to final checkout, ensuring that the status is updated correctly at each step.

+**PaymentService**: Specifically handles the logic for cashless transactions. It invokes the external **Payment Gateway** and updates the system based on the success or failure of the transaction.

+**InventoryService**: A critical supporting service that performs `checkStock()` and `updateStock()` operations. It ensures that when a coffee order is placed, the raw material quantities (beans, milk) are accurately decremented in the database.

3.4.3. Model Layer: Entity & Data Structure

The Model layer defines the objects that are mapped to the physical database tables in the **Database Layer**.

+**Product**: Stores static and dynamic information about items, including price and current stock levels.

+**Order & OrderItem**: Represents a customer's specific purchase. The `Order` entity tracks the overall status and total price, while `OrderItem`

maintains the list of specific products and quantities within that order.

+**Payment & Transaction**: These entities are used to audit financial movements. `Payment` records the status of the billing process, while `Transaction` provides an immutable log of every financial event for daily sales reporting.

3.5.Use Case Diagram

Figure 1: Use Case Diagra m– User Interaction

This diagram presents the primary interactions between external actors and the system, focusing on architecturally significant use cases rather than detailed operational behaviors.

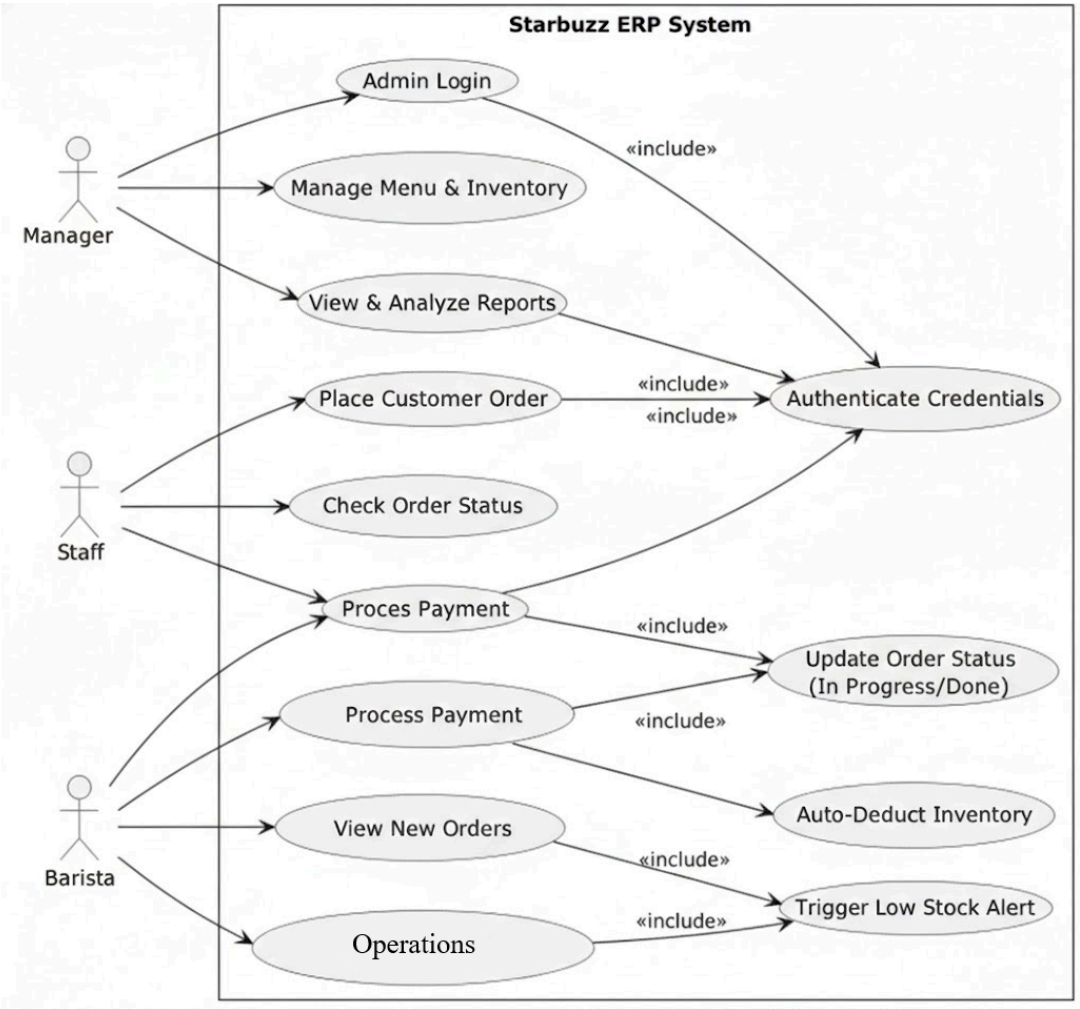


Figure 2: Use Case Diagram – System Functionalities

In the Use Case Diagram – System Functionalities, the system itself is not modeled as an actor. Instead, internal system behaviors such as Identity Authentication, Automatic Stock Deduction, and Waste/Low-Stock Alerts are represented as included use cases using the `<<include>>` relationship. These are automatically executed as part of user-initiated actions, such as processing a payment or updating an inventory batch.

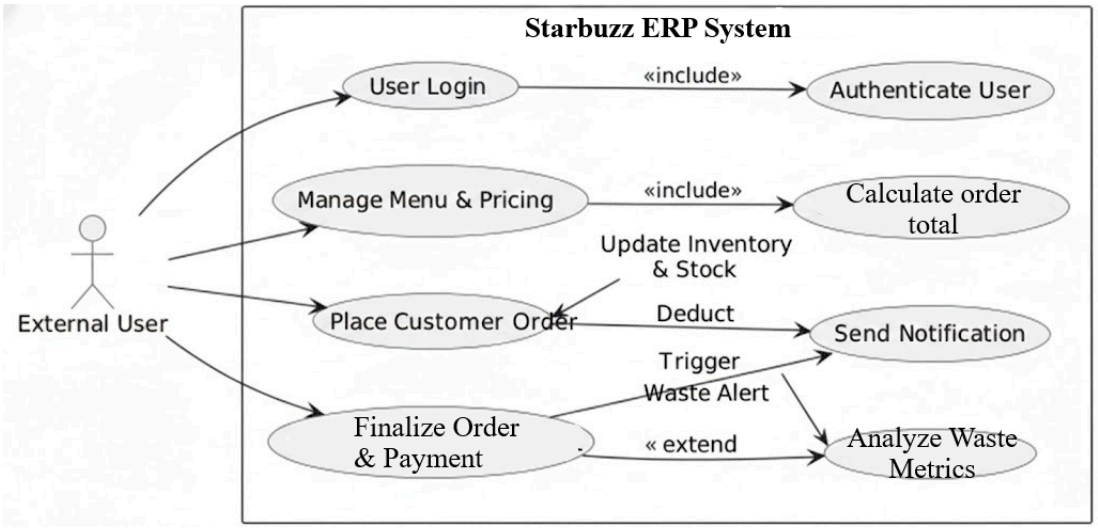
The User actor represents a generalized external entity (encompassing Managers, Staff, and Baristas), while role-specific interactions—such as a Barista monitoring the real-time order queue—are detailed to show how the system maintains real-time synchronization.

+Identity Authentication: Automatically triggered whenever a staff

member logs into the Flutter POS or the React Management Dashboard to ensure secure access.

+Automatic Stock Deduction: Executed immediately upon the completion of a transaction (Process Payment) to ensure the inventory reflects real-time availability.

+Waste & Low-Stock Alerts: Triggered by the Inventory Service when ingredients reach a minimum threshold or approach their expiration date, directly supporting the project's goal to minimize leftover food.



+Specification and Construction of Manager Use Case Group

Overview Description

The **Manager** is the primary actor responsible for overseeing the core operational and logistical data within the **Starbuzz ERP** system. Manager interactions focus on system access control and high-level resource management, including menu configurations, pricing, inventory procurement, and staff oversight. Additionally, the Manager is authorized to monitor business performance by viewing real-time revenue reports and specialized **Waste Analytics** to support the strategic goal of **minimizing leftover food**.

Use Case Specification: UC-M1 Manager Login

Use Case Name	Administrator Login		
Created By	Development Team	Last Updated By	System Officer
Created Date	Jan 5,2026	Last Modified Date	Feb 5,2026
Description	Allows the Manager to authenticate their identity using a username and password to access administrative functions such as inventory control and waste reporting. This function utilizes the centralized **Identity Service**		
Actors	- Manager (Primary Actor) - Identity Service (Authentication System)		
Pre-conditions	1. The Identity Service is operational. 2. The Manager has accessed the Login page on the Web Dashboard or Mobile App. 3. The Manager account exists in the Starbuzz ERP database.		
Post-conditions	Success: 1. The system returns a secure Access Token (JWT). 2. The user is redirected to the **Manager Dashboard**. 3. The management menu displays full administrative functions (Inventory, Reports, Waste Logs).		

	Failure: 1. The user remains on the login page. 2. The system displays a user-friendly error message.
Main Flow	1. **Access Page:** Manager accesses the admin portal URL; the system displays the Login Form. 2. **Enter Credentials:** Manager enters Username and Password and clicks "Login". 3. **Validation:** Frontend performs preliminary validation to ensure fields are not empty. 4. **Send Request:** Frontend sends an HTTP POST request to the Identity Service API. 5. **Authentication:** Identity Service checks the username and compares the submitted password with the stored hash (BCrypt). 6. **Generate Token:** If correct, Identity Service generates a success response containing the JWT and the "Manager" role. 7. **Redirect:** Frontend saves the Token (LocalStorage) and redirects the user to the Manager Dashboard based on the role. 8. **Use Case Ends.**
Alternative Flow	**Step 5a. Incorrect credentials:** → Identity Service fails to find the user or the password hash does not match. → Returns HTTP 401: `{"error": "Invalid credentials"}`. → Frontend displays a red notification: "Incorrect username or password."

	Step 7a. Insufficient privileges: → User logs in with a valid account but the role is "Staff" or "Barista". → Notification: "Access Denied: You do not have permission to access the Manager Dashboard" and redirects to the Staff portal.
Exceptions	**1. Service Connection Loss:** → Identity Service is down. → Frontend receives a connection error. → Displays: "System is under maintenance, please try again later."
Requirements	**1. Security:** Credentials must be encrypted via HTTPS. 2. Performance: Login response time must be < 1 second.

+UC-M2: Manage Inventory & Waste

Use Case Name	Manager Inventory & Waste		
Created By	Development Team	Created By	Development Team
Created Date	January 1, 2026	Created Date	January 1, 2026
Description	Allows Managers to view the list of ingredients and stock levels assigned to their department. It facilitates the entry of waste data (spoiled/leftover items) and the update of stock quantities to ensure accurate inventory.		

Actors	- **Manager** (Primary Actor) - **Inventory Service** (Backend System) - **Product Service** (Verifies ingredient-to-recipe links)
Pre-conditions	1. Manager has logged in successfully 2. The Manager is authorized for the specific store/warehouse 3. The Inventory Period is **OPEN** (not yet locked for monthly reporting)
Post-conditions	Success: 1. Stock levels are updated in the Database. 2. Waste Metrics are automatically re-calculated to track efficiency. Failure: 1. Data remains unchanged. 2. Error message displayed (e.g., "Quantity cannot be negative").
Main Flow	1. **Select Category:** Manager navigates to "Stock Management" and selects a specific category (e.g., Coffee Beans, Dairy). 2. **Input Data:** System displays the current stock list. Manager enters new values: **Actual Quantity**, **Waste Quantity**, and **Expiration Date**. 3. **Validation (Frontend):** The interface checks if the input is numeric and greater than or equal to zero. 4. **Save:** Manager clicks "Update Inventory". Frontend sends an HTTP **PUT** request to `/api/inventory/batch-update`

	5. **Processing:** Inventory Service validates the stock status. Updates records in the Database and triggers a background calculation for **Waste Analytics** (Level 4: Code logic). 6. **Feedback:** System notifies: "Inventory updated successfully". The interface refreshes with real-time data. 7. **Use Case Ends.**
Alternative Flow	**Step 3a. Invalid Input:** Manager enters a negative value. System highlights the cell in red and disables "Save". **Step 5a. Concurrent Edit Conflict:** Two managers attempt to update the same ingredient stock simultaneously. System detects version conflict and notifies: "Data has been modified by another user. Please refresh".
Exceptions	**1. Locked Period:** Manager attempts to save, but the fiscal period is locked. Inventory Service returns **403 Forbidden**. System notifies: "This period is locked and cannot be edited".
Requirements	1. **Data Integrity:** Stock levels must not be negative. 2. **Audit Log:** Every change must be logged (Who, Old Value, New Value) for waste auditing. 3. **Batch Processing:** The API supports batch updates to minimize network latency during large stocktakes.

UC-M4: View Waste Metrics & Inventory Efficiency

Use Case Name	View Waste Metrics & Inventory Efficiency
Created By	Development Team
Created Date	Jan 1, 2026
Description	Allows Managers (and authorized Staff) to view the history of inventory consumption, detailed waste logs per ingredient, and system-calculated efficiency metrics (Waste Ratio, Stock Turnover, Total Savings from waste reduction).
Actors	- **Manager** (Primary Actor) - **Reporting Service** (System)
Pre-conditions	1. User has successfully logged in. 2. Inventory and transaction data exist in the system. 3. Reporting Service is operational.
Post-conditions	Success: 1. Efficiency dashboard and waste transcript are displayed correctly. 2. Waste Ratio indicators match the latest system calculations. Failure: 1. "No data available" message is displayed. 2. Error notification if the service fails.
Main Flow	1. **Access Performance Dashboard:** User selects "Waste & Efficiency" from the dashboard. 2. **Load Data:** System calls **GET** `/api/reports/efficiency-profile` to the Reporting Service. The service aggregates: + **Summary:** Total Ingredients Saved, Waste Ratio (%), Stock Turnover Rate. + **Details:** List of ingredients with their respective waste logs grouped by Date/Shift. 3. **Display Detailed Waste Log:** System displays a table of ingredients. Columns: Ingredient Name, Unit, Amount Discarded, Reason (Expired/Spilled), Cost Impact.

	4. **Filter (Optional):** User filters by "Morning Shift - Jan 30". System updates the view to show only that shift's metrics. 5. **Use Case Ends**
Alternative Flow	Step 2a. Insufficient Data (New Branch): → The store has not recorded enough transactions yet. → System displays: "Insufficient data for efficiency calculation" with Metrics = 0.0. Step 2b. Role-Based View (Staff): → User is a Staff member, not a Manager. → System hides sensitive cost data, showing only physical waste amounts.
Exceptions	**1. Calculation Error:** Data corruption causes a division by zero in the Waste Ratio calculation. System handles it gracefully, displaying "N/A". **2. Display Error:** Ingredient names are missing due to a sync issue with the **Product Service**. System displays "Unknown Item [ID]" temporarily.
Requirements	1. **Accuracy:** Metrics must be calculated to 2 decimal places. 2. **Real-time:** The report must reflect all transactions finalized in the last 5 minutes.

Use Case Specification: QR Code Payment Integration:

ID and Name	UC-QR: QR Code Payment Processing
Created By	Group 06 / Date created: February 6, 2026
Primary Actor	Staff, Customer
Description	Allows customers to pay for their orders by scanning a dynamic QR code generated by the system.

Trigger	Staff clicks the "QR Payment" button on the order checkout screen.
Preconditions	PRE-1: The order is in "Pending Payment" status. PRE-2: The Payment Service and API Gateway are operational.
Postconditions	POS-1: Order status is updated to "Paid" in the database. POS-2: A `PaymentConfirmed` event is published to the notification queue
Normal Flow	1. Staff selects the order and initiates payment. 2. API Gateway routes the request to the Payment Service. 3. Payment Service retrieves order details (Amount, OrderID) from the Order Service. 4. Payment Service generates a dynamic QR code via a 3rd-party banking API. 5. The QR code is displayed on the customer-facing interface. 6. Upon successful transaction, the system updates the database and displays "Payment Successful".
Exceptions	5a. Transaction Timeout: System notifies staff that the QR code has expired and offers to regenerate it
Priority	High

Use Case Specification: Sales Statistics & Analytics

ID and Name	UC-STAT: Sales Statistics & Reporting
Created By	Group 06 / Date created: February 6, 2026
Primary Actor	Administrator (Admin)
Description	Provides a comprehensive overview of revenue by day, month, or product category for the coffee shop.

Trigger	Admin accesses the "Reports & Statistics" menu.
Preconditions	PRE-1: Admin is successfully authenticated. PRE-2: Transaction data exists in the system (Orders/Payments)
Postconditions	POS-1: Statistical data is visualized via tables or charts on the dashboard.
Normal Flow	
Exceptions	
Priority	

4. TESTING & VERIFICATION

4.1 Testing Strategy

Due to the distributed nature of the Microservices architecture, the project team implemented a multi-layered testing strategy. This approach ensures that each service, such as Order, Inventory, and Identity, functions correctly in isolation before being integrated into the full system.

4.1.1 Unit Testing

This phase focuses on testing the smallest components within each Microservice to ensure internal data processing logic is accurate:

+**Model Testing:** Verifies entity classes such as Product, InventoryItem, and Order within the models.py file. This ensures that data initialization and JSON serialization (to_dict) are performed correctly.

+**Repository Testing:** Validates data access methods within repository.py. It ensures that SQL commands for inserting orders or updating stock levels interact correctly with the MySQL database and handle connection exceptions gracefully.

4.1.2 Integration Testing

Integration testing focuses on the interactions between services via APIs and

real-time communication protocols:

+**API Testing:** Utilizes tools like Postman to issue HTTP requests (GET, POST, PUT) to various endpoints. The objective is to verify HTTP status codes (200, 201, 400, 404) and ensure the returned JSON structures match the technical specifications.

+**Inter-service Communication:** Tests the asynchronous data flow between the **Order Service** and the **Inventory Service**. This confirms that once a payment is confirmed, the system triggers the stock deduction logic without causing latency in the user interface.

4.2 Deployment Configuration

The system is designed for flexible deployment in a local environment (Localhost) with standardized security and port management.

4.2.1 Environment Variables (.env)

To maintain security and allow for easy configuration changes across different environments (Development/Production), sensitive information such as database credentials is managed through a .env file.

+**Configuration Parameters:**

-DB_HOST: Database server address.

-DB_NAME: starbuzz_erp.

-DB_PASS: Secured via hashing and environment-level encryption.

4.2.2 Port Mapping

In the Microservices architecture, each service is assigned a unique port to prevent system conflicts:

Service Name	Port	Primary Function
Identity Service	5001	Authentication & Authorization (JWT).
Product Service	5002	Menu and pricing management.
Order Service	5003	Transaction processing & real-time WebSocket communication.
Inventory Service	5004	Stock tracking & waste alerts.
Reportin g Service	5005	Revenue & waste analytics reporting.

4.3 End-to-End (E2E) Test Scenarios

This section describes a comprehensive business process simulation, moving from the **Flutter** mobile interface to the backend database.

Scenario: Sales Process & Intelligent Stock Update

- **Objective:** Verify data consistency across the flow: Identity $\rightarrow$ Product $\rightarrow$ Order $\rightarrow$ Inventory.

1. **Login:** Staff members access the system via the Flutter app. The system returns a JWT for role-based authentication.
2. **Ordering:** Staff selects a "Starbuzz Coffee" item. The system calls the **Product Service** to retrieve recipe specifications and pricing.
3. **Preparation:** The order appears instantly on the Barista's queue through

a WebSocket connection (Socket.IO).

4. **Payment & Deduction:** Upon payment, the **Order Service** notifies the **Inventory Service** to automatically deduct the corresponding raw materials.
5. **Waste Mitigation:** If an ingredient (e.g., fresh milk) is nearing its expiration date or stock falls below a minimum threshold, the system triggers an alert to **minimize leftover food**.

Verification Results: The system responded to all read/write operations within < 200ms. The asynchronous background processing ensured that inventory updates and alerts did not hinder the customer's checkout experience.

5. Conclusion & Reflection

5.1 Lessons Learned

Through the implementation of the **Starbuzz ERP (Coffee Shop Management System)** based on **Microservices Architecture**, the project team has gained valuable practical experience that bridges the gap between theoretical software architecture and real-world business operation.

+**Architecture Evolution:** The project helped the team clearly understand the trade-offs between Monolithic and Microservices architectures. While Microservices offer significant advantages in independent deployment and fault isolation—essential for a high-traffic retail environment—they introduce complexity in service coordination and inter-service communication.

+**Service Boundaries:** Designing clear boundaries for services like **Identity, Order, Product, and Inventory** proved to be a critical architectural decision that directly impacted the system's maintainability.

+**Distributed System Challenges:** The team gained hands-on experience with ensuring data consistency and fault tolerance, particularly in the **Inventory Service**. Scenarios such as high-volume

ordering during morning peak hours required careful handling to ensure accurate stock deductions and to avoid "out-of-stock" errors.

+**Mission-Driven Design:** The project reinforced the importance of translating specific business goals, such as the mission to **minimize leftover food**, into concrete architectural features like automated expiry alerts and waste analytics.

+**Non-Functional Requirements (NFRs):** Performance, security, and reliability were not treated as afterthoughts but were explicitly translated into concrete design decisions, such as **JWT-based authentication**, **Role-Based Access Control (RBAC)**, and **Socket.IO** for real-time order broadcasting.

+**Collaborative Standards:** Maintaining consistent API standards and coding conventions required effective communication. The experience emphasized the value of clear interface contracts and structured testing to support teamwork in a multi-platform environment involving Web and **Flutter** applications.

5.2 Future Improvements

Although **Starbuzz ERP** has achieved its primary objectives, several areas can be enhanced in future iterations to further optimize shop operations.

+**Message Broker Integration:** From an architectural perspective, the system could benefit from introducing a message broker (such as **RabbitMQ** or **Kafka**) to replace basic threading for asynchronous tasks. This would improve reliability and scalability for background processes like supplier notifications and large inventory data imports.

+**Advanced Orchestration:** In terms of deployment, containerization using **Docker** and orchestration with **Kubernetes** would allow services to scale independently based on workload, particularly useful during seasonal sales or peak hours.

+**AI-Powered Waste Reduction:** From a functional standpoint, the

system can be extended to include **AI/ML-based predictive analytics**. By analyzing historical sales data, the system could predict daily demand more accurately, allowing for even tighter control over raw material procurement to further **minimize leftovers**.

+**Expanded Ecosystem:** Future versions could support additional features such as **online payment integration**, a customer loyalty mobile app, and integration with external delivery platforms via the API Gateway.

In conclusion, this project serves as a solid foundation for applying **Microservices Architecture** to a real-world F&B management problem. The lessons learned and proposed improvements provide a clear roadmap for the future development of **Starbuzz ERP**, ensuring it remains a modern, distributed, and sustainable software solution.