Coffee Shop Manager

System Requirements Specification
V4

Introduction

Small coffee shops often struggle with subpar or fragmented tools that don't fit their use case. This project aims to solve that by designing an integrated system that includes:

- Point-of-sale (POS)
- Inventory management
- Multiple roles (Customer, Barista, Manager, Financial Manager)
- Mobile application for online ordering

User Characteristics

Customer

- Walk-in or online user with basic to moderate tech skills
- Uses mobile or web app to place orders, manage account, track loyalty points
- Expects fast, responsive UI and clear feedback
- High priority on usability and convenience

Barista

- Oversees inventory, staff schedules, and operations
- Uses admin dashboard tools for real-time monitoring
- Elevated access to view/edit product data and run reports
- Good system proficiency but may need training

Manager

- Monitors sales and inventory
- Can assign employee roles and permissions
- Requires timely, accurate reports

Financial Manager

- Less frequent user, mainly for analytics and reporting
- Needs access to daily/monthly performance metrics
- Works with visualized data and CSV exports
- High need for data accuracy and availability

User Stories

Customer

- Register and create an account
- Log in to view past orders, loyalty points, place orders
- Reset password independently
- Access and update account profile
- Browse full menu and descriptions
- Order coffee via mobile to skip the queue

Barista

- See incoming orders in a queue

- Update order status for real-time feedback
- Notify customers when order is ready
- Monitor stock levels and mark items “out of stock”
- Print receipts after payment
- Apply loyalty points discounts

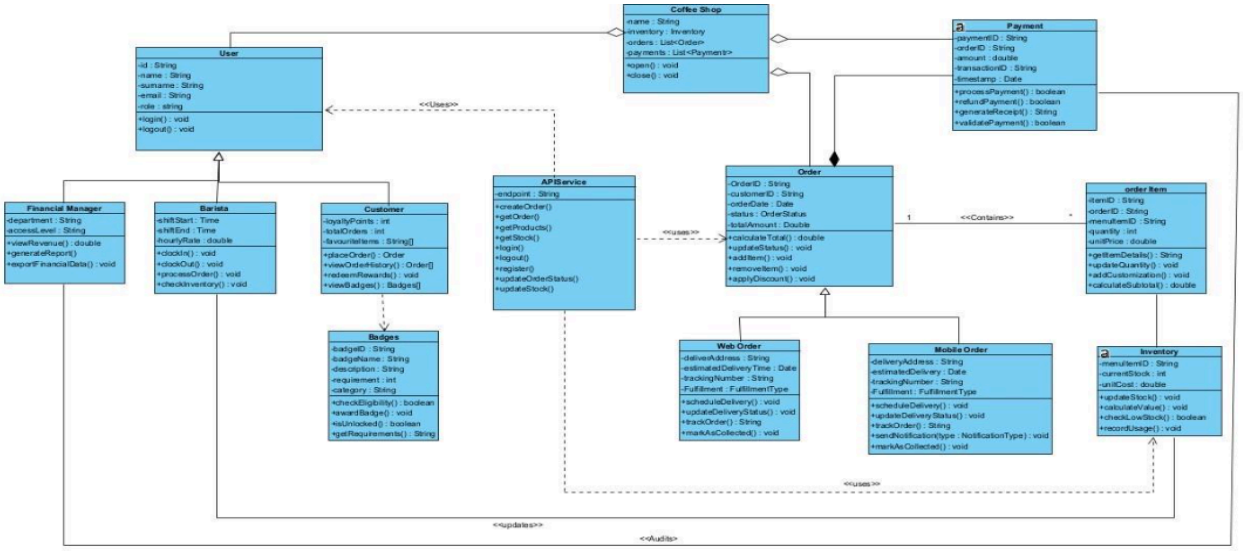
Manager

- View inventory levels for restocking
- View daily sales reports to analyze performance

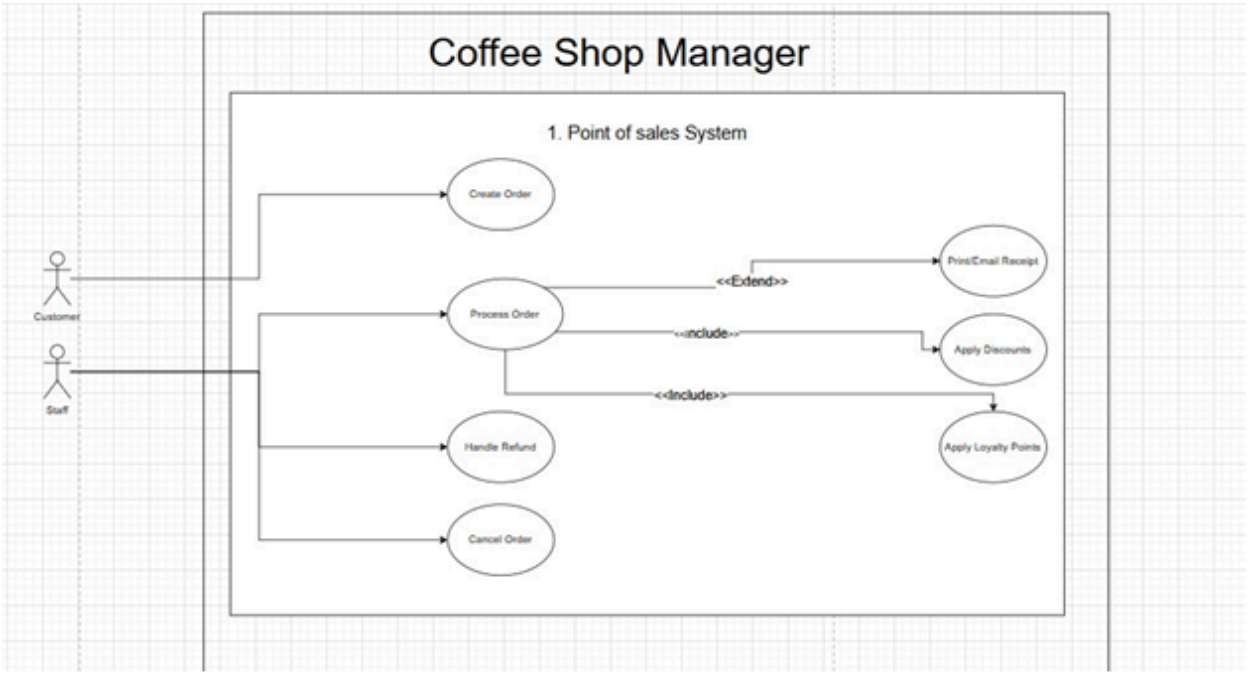
Financial Manager

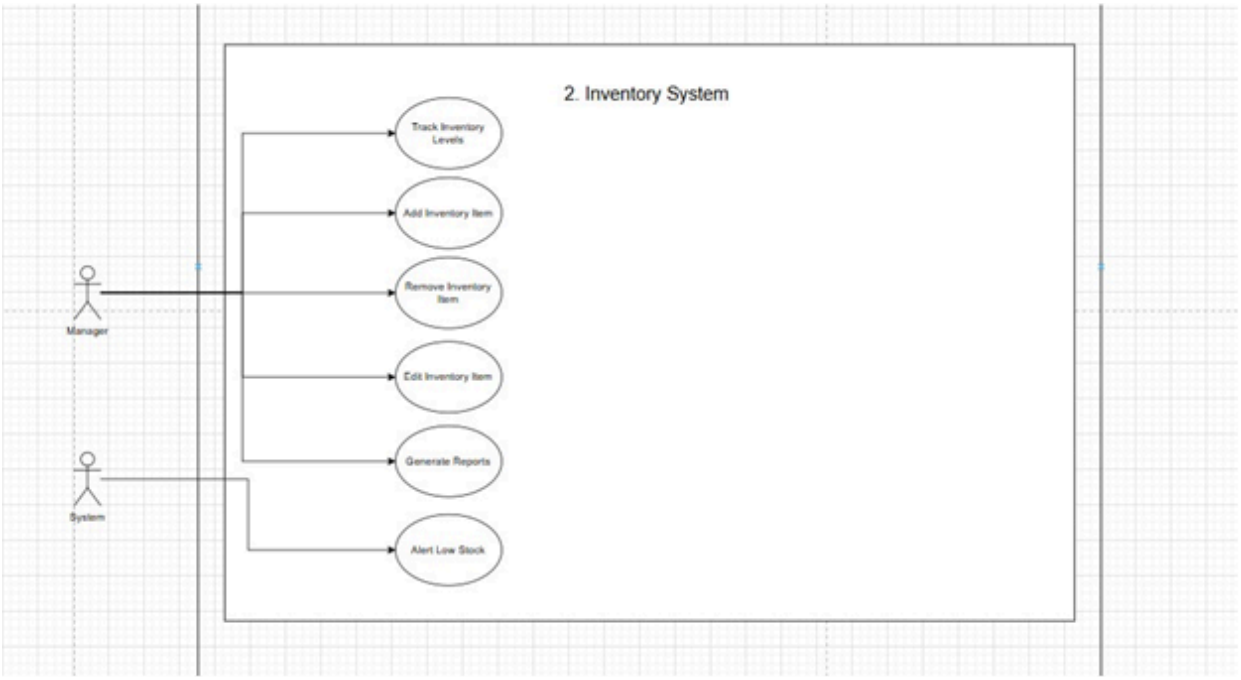
- View daily sales reports to track revenue
- Track expenses to maximize profit
- Track labor costs per employee

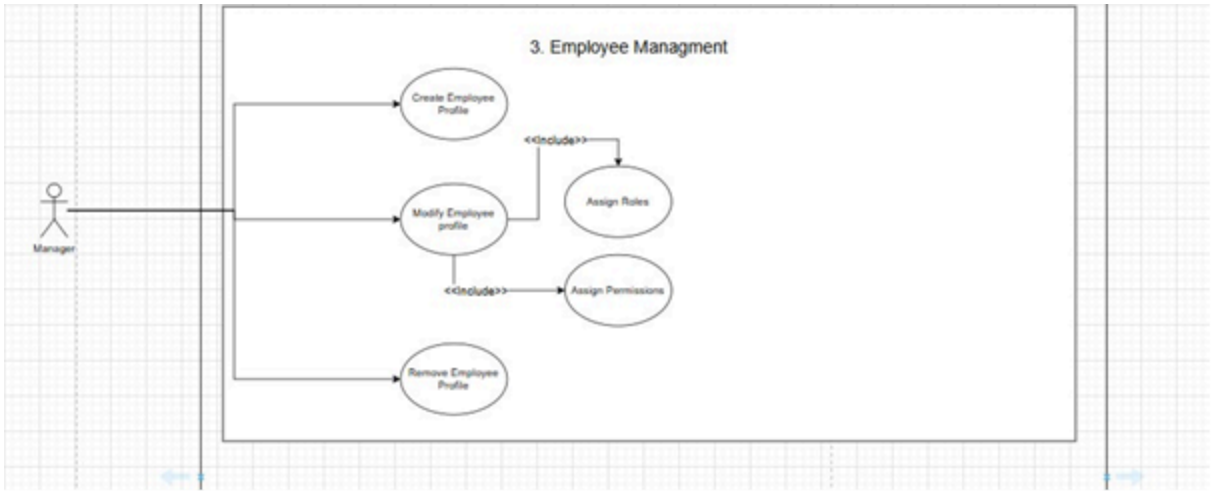
Domain Model Diagram

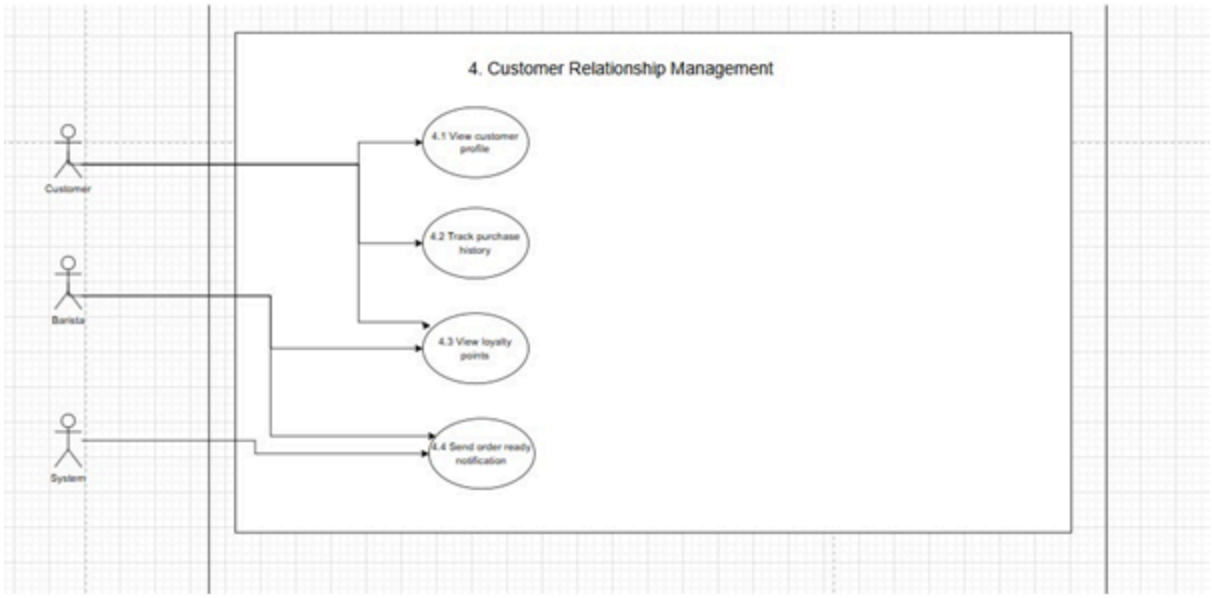


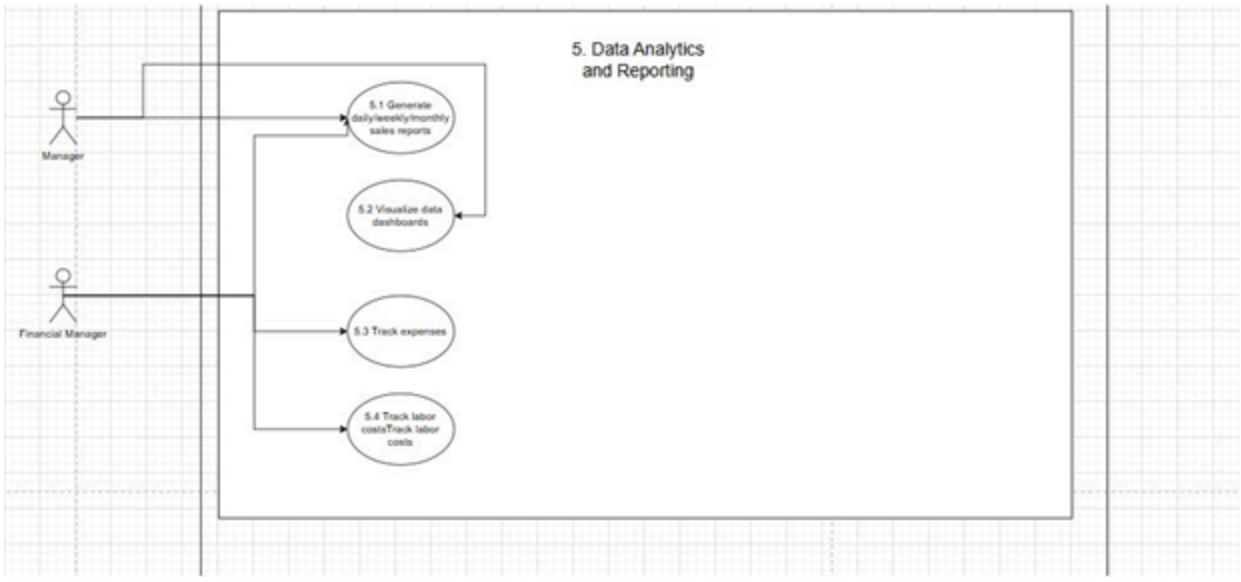
Use Case Diagram

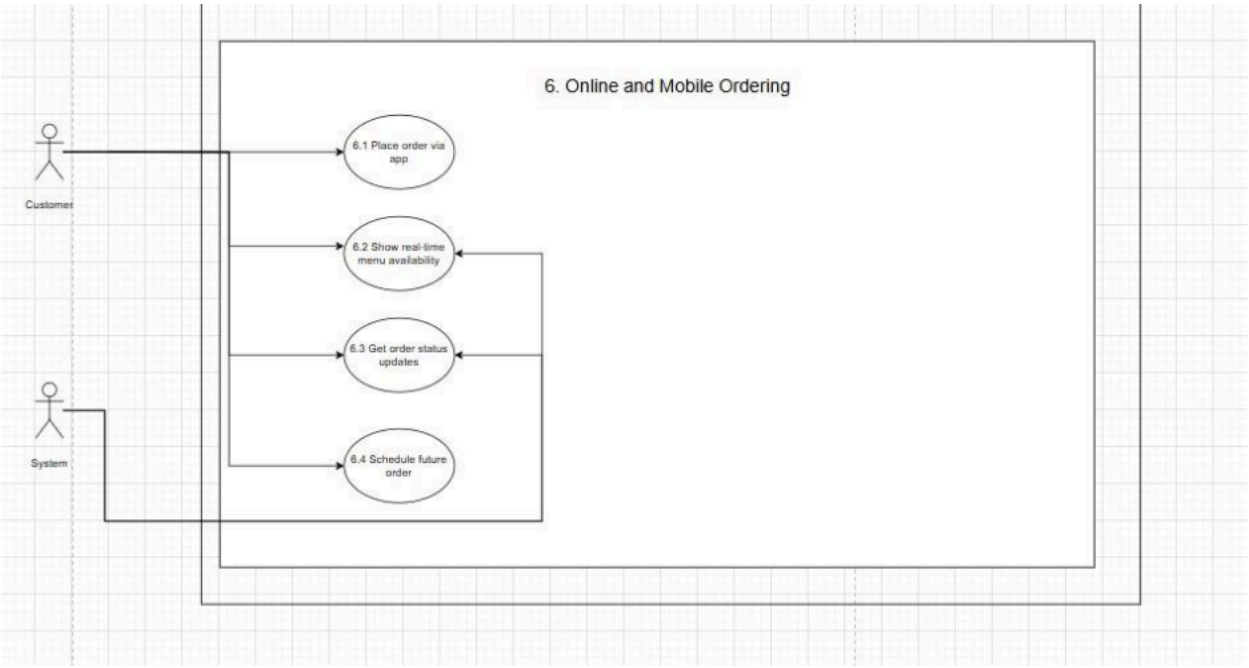

Coffee Shop Manager
1. Point of sales System
Create Order
Process Order
Handle Refund
Cancel Order
Print/Email Receipt
Apply Discounts
Apply Loyalty Points
<<Extend>>
<<include>>
<<include>>
Customer
Staff


2. Inventory System
Track Inventory Levels
Add Inventory Item
Remove Inventory Item
Edit Inventory Item
Generate Reports
Alert Low Stock
Manager
System


3. Employee Managment
Create Employee Profile
<<Include>>
Assign Roles
Manager
Modify Employee profile
<<Include>>
Assign Permissions
Remove Employee Profile


4. Customer Relationship Management
Customer
4.1 View customer profile
4.2 Track purchase history
Barista
4.3 View loyalty points
System
4.4 Send order ready notification


5. Data Analytics and Reporting
Manager
5.1 Generate daily/weekly/monthly sales reports
5.2 Visualize data dashboards
Financial Manager
5.3 Track expenses
5.4 Track labor costsTrack labor costs



Functional Requirements

Point of Sale (POS)

- Create and process orders
- Apply discounts and loyalty points
- Print or email receipts
- Handle refunds and cancellations

Inventory Management

- Track inventory in real time
- Alert managers on low stock
- Add/edit/remove inventory items
- Generate usage and wastage reports
- Support manual and bulk stock updates

Employee Management

- Create, modify, remove employee profiles
- Assign roles and permissions

Customer Relationship Management

- Store profiles and preferences
- Track purchase history
- Display loyalty points
- Notify customers when orders are ready

Data Analytics & Reporting

- Daily, weekly, monthly sales reports
- Visualized dashboards

Online & Mobile Ordering

- Place orders via web or mobile
- Show real-time menu availability
- Real-time order status updates
- Schedule orders

Quality Requirements

1. Security
    - The system must enforce JWT authentication and role based access control to ensure that only authorized users can access resources.
    - All API endpoints are protected by JWT middleware that validates tokens on every request.
    - Secure user password storage using Supabase's built in authentication with bcrypt hashing.
    - All API traffic is served over HTTPS through Render hosting.
    - Input validation and sanitization on all endpoints to prevent injection attacks.
2. Scalability
    - The system must support throughput of 200 orders per minute with horizontal scaling abilities.
    - System configured to support multiple concurrent requests in Render with adjusted resources.
    - Database query optimization.
    - Load testing using JMeter to validate 200 orders per minute capacity.
3. Availability
    - The system is expected to achieve an uptime of 99.5% uptime ensuring minimal downtime and continuous service availability

4. Performance
    - The system must achieve a 95th percentile API latency of less than 800ms under normal conditions.
    - Database query optimization while monitoring response times.

5. Reliability
    - The system must ensure that all transactions are only processed once and recovery mechanisms without data loss

- Database constraints to prevent duplicate orders. Ex unique order IDs.
- Automated daily integrity checks.

6. Maintainability
    - The system maintains clean, well documented code architecture supporting efficient updates and bug fixes.
    - Maintain comprehensive up to data API documentation.
    - Implement modular architecture with clear separation of concerns. Controllers must handle HTTP requests and services contain business logic.

7. Usability
    - The mobile application must allow users to complete an order in less than 6 taps for an intuitive experience.
    - Follow UI/UX guidelines ensuring mobile app is tested against task completion metrics.
8. Testability
    - The system must achieve at least 90% unit test coverage on all critical modules.
    - Automated unit tests run via GitHub Actions before deployment.
    - Generate code coverage reports and reviewed in CI/CD pipelines.

Quality Requirements testing

Overview

To ensure that the coffee shop manager system meets its defined quality requirements a comprehensive non functional testing was conducted using K6 load testing framework. Each quality requirement with metrics was tested under realistic load conditions to ensure the system performs as specified.

Testing Environment

Load testing tool: K6
Test configuration:

- **Virtual Users**
- **Test duration:** 30 seconds to 3 minutes.
- **Network conditions**: Local testing environment because of the limit on our hosting service.
- **Ramp up strategy:** Gradual increase with each stage increasing the number of users.

Quality requirements test results

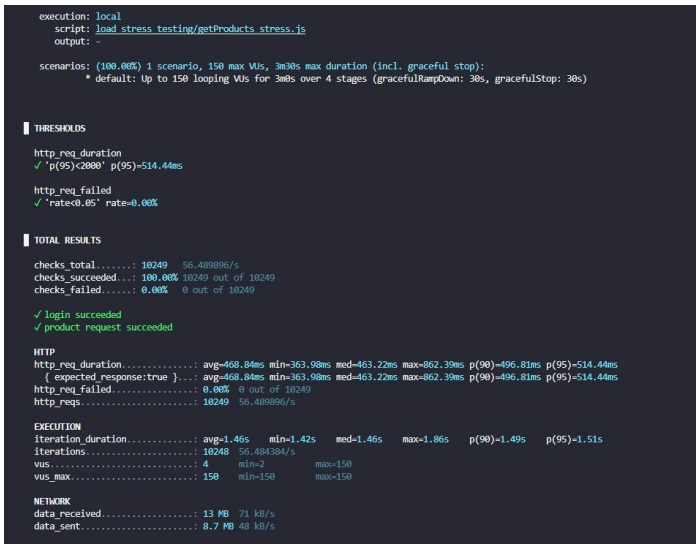
1. **Performance testing**
   Quality requirement: The system must achieve a 95th percentile API latency of less than 300ms under normal conditions.

   Test 1: Get products endpoint (stress load)

   Test configuration:
   - Script: getProducts_stress.js
   - Load:150 max virtual users over 3 minutes
   - Total requests: 10,249 requests

   Results
   - Average response time: 468.84ms
   - Success rate: 100% (10249 / 10249)
   - Throughput: 56.48989/s

Analysis: The 95th percentile latency was below the 2000ms requirement. All tests pass indicating very good performance.



2. Get products endpoint (normal load)

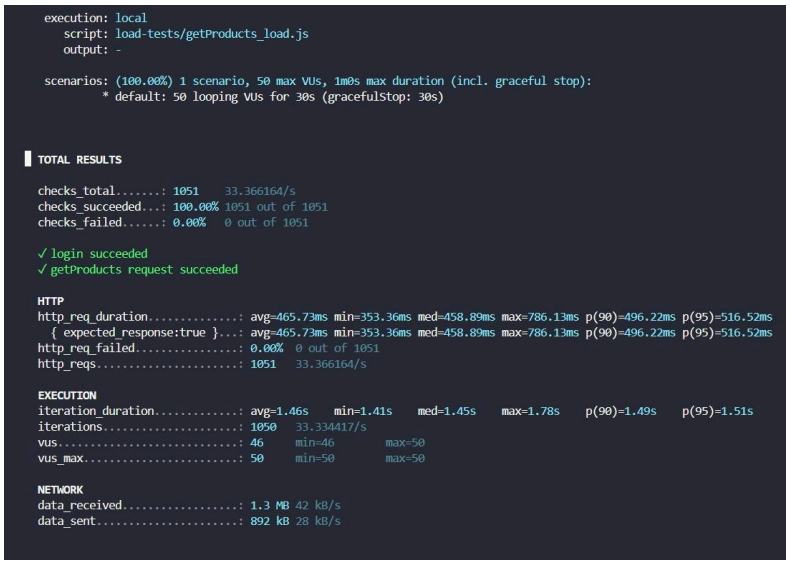
   Test configuration:
   - Script: getProducts_load.js

- Load: 50 max virtual users over 30 seconds
- Total requests: 1051 requests

Results:

- Average response time: 465.73ms
- Success rate: 100% (1051 / 1051)
- Throughput 33.366164/s

Analysis: Under normal load conditions, the 95th percentile (516.52ms) comfortably meets the 800ms requirement, demonstrating stable performance.



Test 3: Get orders endpoint (Normal load)

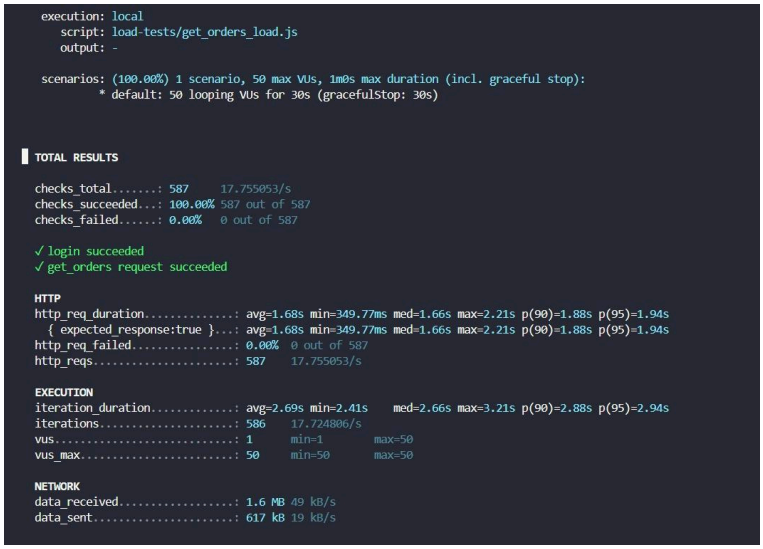
Test configuration

- Script: get_orders_load.js
- Load: 50 max virtual users over 30 seconds
- Total requests: 587 requests

Results:

- 95th percentile: 1.94s
- Average response time: 1.68s
- Success rate: 100% (587 / 587)
- Throughput: 17.755053/s

Analysis: The orders endpoint shows concerning performance with 95th percentile of 1.94s, exceeding the 800ms requirement by 1.14s. This endpoint requires optimization.



Test 4: Create order endpoint (Normal load)

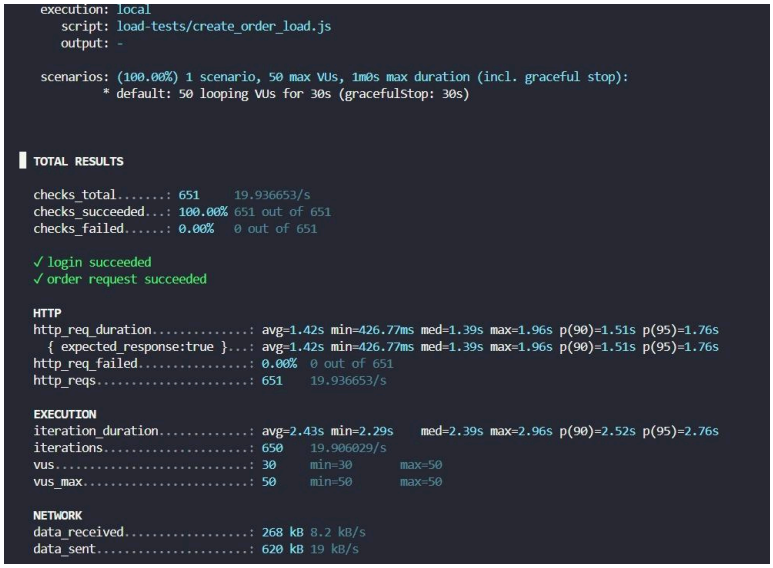
Test configuration:

- Script: create_order_load.js
- Load: 50 max Virtual users over 30 seconds.
- Total requests: 651 requests

Results:

- 95th percentile response time: 1.76s
- Average response time: 1.42s
- Success rate: 100% (651 / 651)
- Throughput: 19.936653/s

Analysis: Order creation endpoint exceeds the 800ms requirement with 95th percentile of 1.76s. This critical operation needs performance improvements.



2. Scalability testing

Quality requirement: The system must support throughput of 200 orders per minute.

Test 5: Order creation stress test

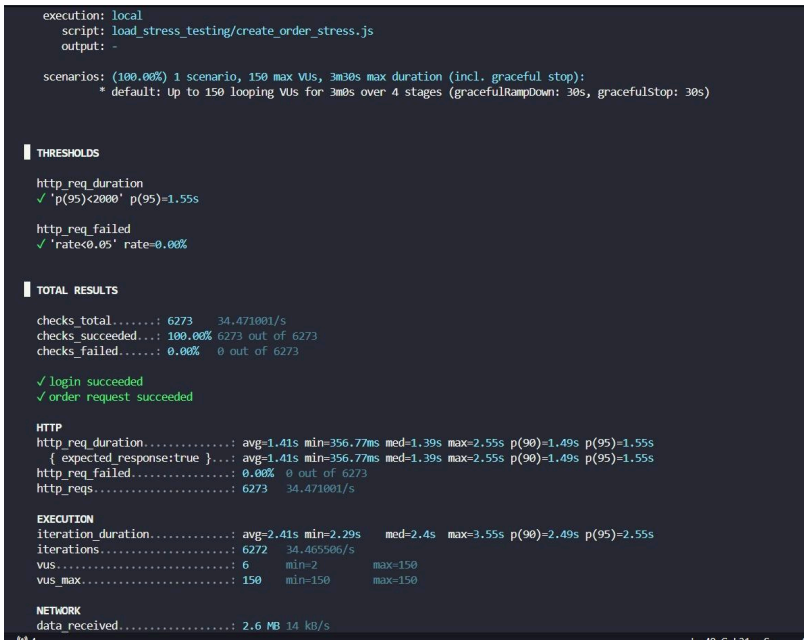
Test configuration

- Script: create_order_stress.js
- Load: 150 max virtual users over 3 minutes
- Total requests: 6,273 requests

Results:

- Throughput: 34.471001/s
- 95th percentile response time: 1.55s
- Success rate: 100% (6273 / 6273)

Analysis: The system exceeds the throughput requirement handling 2068 orders per minute which is over 10 times the required orders per minute.



3. **Reliability testing**

Quality requirement: The system must ensure that all transactions are processed exactly once with a 0% failure rate under normal conditions.

Reliability Test Results (Testing all All Endpoints)

Get Products Tests:

- Stress Test: 0% failure rate (10,249/10,249 successful)
- Load Test: 0% failure rate (1,051/1,051 successful)

Get Orders Test:

- Load Test: 0% failure rate (587/587 successful)

Create Order Tests:

- Stress Test: 0% failure rate (6,273/6,273 successful)
- Load Test: 0% failure rate (651/651 successful)

Get Orders (Alternative Test):

- Load Test: 0% failure rate (1,205/1,205 successful)

Analysis: All endpoints demonstrated 100% reliability with zero failed requests across all test scenarios meeting the reliability requirement.

4. **Availability testing**

Quality requirement: The system is expected to achieve a 99.5% uptime

Test results:

- System uptime during testing: 100%
- No interruptions: All tests completed successfully
- Zero downtime: No failed requests due to service unavailable or offline.

Analysis: During the testing period the system maintained a 100% availability exceeding the 99.5% requirement.

Architectural Requirements

Architecture Style

1. **3 layer architecture**

   The system is organized into distinct layers. Each layer with specific responsibilities and dependencies.

   Layer structure:
   - Presentation layer: User interface and user interaction management.
   - Business logic layer: Core application functionality and business rules
   - Data layer: Data persistence and retrieval operations.

Justification through Quality attributes

Maintainability: Clear separation allows developers to modify one layer without affecting others. Changes to user interface components do not impact business logic and database modifications are isolated from presentation.

Testability: Each layer can be tested independently. Integration testing can target specific layer interactions.

Modifiability: New features can be added by extending the appropriate layer without changing the structure of the entire system. For example interface changes do not impact the business logic layer.

2. **Client server architecture**

   The system distributes functionality between client and server processes and communicates through protocols.

   Components:

   - **Client processes:** Handle user interface display and local user interactions.
   - **Server processes:** Manage business logic execution and data operations.
   - **Communication:** Request response mechanism between the client and server.

   Justification through quality attributes

   - **Scalability:** Multiple clients can connect to the same server and server resources can be scaled depending on demand. Load can be distributed without affecting client functionality.
   - **Security:** Centralized business logic and data access control on the server. Sensitive operations are performed on the secure server architecture rather than client devices which may potentially be compromised.
   - **Performance:** Tasks are distributed between client and server ensuring efficient resource utilization and enabling effective load balancing under high workloads.

3. **Model view controller (MVC)**
   The coffee shop manager is structured into three interconnected components. This division isolates concerns and ensures responsibilities are clearly defined.

Components:

- **Model:** Manages the core data, business logic and rules of the application.

- **View:** Handles the presentation of data to the user. Renders user interface elements and updates based on model changes.
- **Controller:** Acts as the intermediate between the view and model, processing user input, triggering business logic and updating both components if any changes occur.

Justification through quality attributes:

- **Maintainability:** The MVC pattern enforces clear boundaries. Modifications can be made to the UI without altering the underlying logic or changing the business logic without impacting the view.
- **Testability:** Each component can be tested independently.
- **Reusability:** Views can be reused across multiple different contexts while the controller remains unchanged. Additional interfaces for example web and mobile can be integrated by reusing models and controllers with different views.

Architectural Patterns

The system uses the following design patterns to improve maintainability, scalability, and flexibility:

- Singleton: ensures a single instance of the database connection across the app.
- Factory: handles creation of objects like menu items, orders, and users.
- Observer: used for notifications (e.g., order status updates).
- Strategy: implements different discount and loyalty strategies.
- MVC (Model-View-Controller): organises frontend code for React Native and Next.js to separate data (Model), UI (View), and logic/controllers (Controller).
- Client-Server: separates presentation (frontend) from business logic and data (backend).

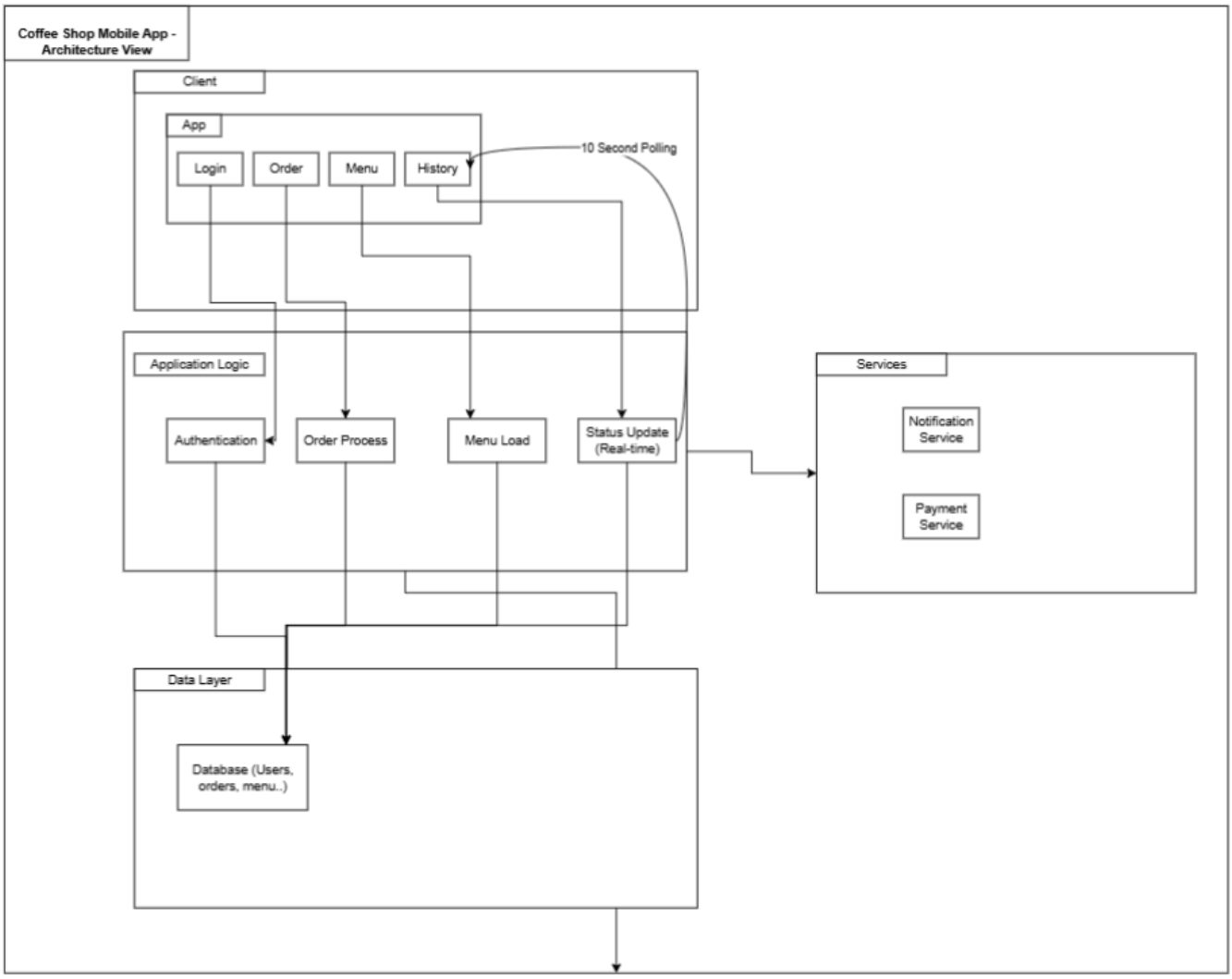
Extended Patterns for Demo 4

- Certain components (like order processing or payment services) may be implemented as microservices to show modular scalability and

independent deployment.

Justification

- 3-tier architecture clearly separates concerns: UI, logic, and data.
- Patterns enhance code reuse, flexibility, and maintainability.
- Client-server approach ensures the system can easily support multiple clients (mobile and web).
- Microservices (optional for Demo 3) demonstrate the potential for scaling individual services independently



Constraints

- Must use PostgreSQL
- Must use React Native for mobile
- Must run on Node.js, deployable via Docker
- Only open-source libraries
- Web & mobile share backend API

Technology Requirements

Component	Technology
Web Frontend	Next.js (React, TypeScript)
Mobile App	React Native
Backend API	Node.js + Express
Database	PostgreSQL + Drizzle ORM

Authentication	JWT + Role-based access
Styling	Tailwind CSS
DevOps / CI/CD	GitHub Actions, Docker
Testing	Jest, React Testing Library

Deployment Model

- Environment: Cloud-hosted containers
- Topology: Multi-tier, containerized microservices (Orders, Inventory, Loyalty, Analytics)
- Tools: Docker, GitHub Actions CI/CD

Service Contracts (API Spec)

The coffee shop manager consists of several major components: Web front end, mobile app, backend API and database.

Base URL's

- Development: http://localhost:5000
- Hosted: https://api.diekoffieblik.co.za

API conventions

- All requests and responses are in JSON format
- REST/HTTPS protocols
- Versioning strategy
- HTTP status codes used:
    - 200 OK - Successful GET or PUT request
    - 201 CREATED - Successful resource creation
    - 400 bad request - Client-side error
    - 404 Not Found - Requested resource does not exist
    - 500 Internal Server Error - server error

Authentication / Authorization

- Authentication
    - All API requests must include a valid JSON Web Token (JWT) stored in cookies
    - Tokens are issued after successful login via the /login endpoint
    - Tokens are valid for 60 minutes and can be refreshed to restore the session

- User roles and permissions
    - Admin: Has unlimited to all data
    - Barista: Has access to all order data
    - User: Has access to only their own personal data (i.e. orders, profile)

- Password and security
    - Authentication is managed by Supabase Auth which handles user login and registration.
    - Supabase ensures that passwords are stored using secure salting and hashing algorithms.
    - Passwords must have a minimum length of 8 characters

API endpoints

1. Users

   1.1 Sign up endpoint

   Purpose: Create a new user.
   Method: POST

Request Body

The request body must be in JSON format and should contain the following parameters:

- `username` (string)
  - The desired username for the new account.
- `email` (string)
  - The email address associated with the account.
- `password` (string)
  - The password for the account.

Request body example

json

```
{
  "username": "testuser",
  "email": "testing@coffee.com",
  "password": "testing"
}
```

Response

On a successful request, the server responds with a status code of `201 Created` and a JSON object containing the following structure:

- `success` (boolean):
  - Indicates whether the signup was successful.
- `message` (string):
  - A message providing additional information.
- `role` (string):
  - Contains details about the newly created user.

Response example

json

```
{
  "success": true,
  "message": "User registered successfully",
  "role": "user"
}
```

1.2 Login endpoint

Method: POST
Path: /login

Request Body

The request body must be in JSON format and should contain the following parameters:

- email (string)
    - The email address associated with the account.
- password (string)
    - The password for the account.

Example request body

json

```
{
  "email": "user@coffee.com",
  "password": "user"
}
```

Response

On a successful request, the server responds with a status code of `200 OK` and a JSON object containing the following structure:

- `success` (boolean):
    - Indicates whether the signup was successful.
- `username` (string):
    - A message providing additional information.
- `role` (string):
    - Contains details about the newly created user.

Example response

json

```
{
  "success": true,
  "username": "anon",
  "role": "user"
}
```

1.3 Delete endpoint

- Method: DELETE
- PATH: `/user/{id}`
    - `{id}` : id of user to be deleted

Request Body

- None

Response

On a successful request, the server responds with a status code of `200 OK` and a JSON object containing the following structure:

- `success` (boolean):
    - Indicates whether the delete was successful.

- `message` (string):
    - A message providing additional information.

json

```
{
  "success": true,
  "message": "User {id} deleted successfully."
}
```

2. Users

2.1 Create Product

Purpose: Creates a new product

Method: POST

PATH: /product

Request body

The request body must be in JSON format and should contain the following parameters:

- `name` (string)
    - Name of product.
- `description` (string)
    - Description of product.
- `price` (float)
    - Price of product.
- `stock_quantity` (int)
    - Not required.
    - Amount of items int stock.
- `stock_items` (array)
    - Items used to make product.
    - `item` (string)
        - Name or id of stock item.
    - `quantity` (int)
        - Amount of stock item used.

Request body example

```json
{
  "name": "New Drink",
  "description": "A refreshing new beverage.",
  "price": 40.00,
  "stock_quantity": 50,
  "stock_items": [
    { "item": "Coffee Beans", "quantity": 2 },
    { "item": "Sugar", "quantity": 3 }
  ]
}
```

Response

On a successful request, the server responds with a status code of `201 Created` and a JSON object containing the following structure:

- `success` (boolean):
    - Indicates whether the signup was successful.
- `message` (string):
    - A message providing additional information.
- `product_id` (uid):
    - Id of created product.

2.2 Get products

Purpose: Gets products

Method: GET

PATH: /product or /product/{id}

Request body

None.

ID can be provided in path to return specific item.

Response

On a successful request, the server responds with a status code of `200 OK` and a JSON object containing the following structure:

- `success` (array):
    - Contains all returned products.
    - `id` (uid)
    - `name` (string)
    - `description` (string)
    - `price` (float)
    - `stock_quantity` (int)

Example Response:

Contains two items.

```
[
    {
        "id": "d9fad8dc-3d4a-4a65-9e28-48b92c778be9",
        "name": "Cappuccino",
        "description": "A rich espresso-based drink topped with steamed milk and foam.",
        "price": 32,
        "stock_quantity": 10
    },
    {
        "id": "a02df74b-7103-4e8e-9d43-b3b305b2fd18",
        "name": "Latte",
        "description": "Espresso with steamed milk and a light layer of foam.",
        "price": 35,
        "stock_quantity": 20
    }
]
```

2.3 Get products with stock

-

Purpose: Gets products with stock items / ingredients

Method: GET

PATH: /product/stock or /product/stock{id}

Request Body

None.

ID can be provided in path to return specific item.

Response

On a successful request, the server responds with a status code of `200 OK` and a JSON object containing the following structure:

- `success` (array):
    - Contains all returned products.
    - `id` (uid)
    - `name` (string)
    - `description` (string)
    - `price` (float)
    - `stock_quantity` (int)
    - `ingredients` (array)

Example Response:

Contains two items.


```
[
    {
        "id": "d9fad8dc-3d4a-4a65-9e28-48b92c778be9",
        "name": "Cappuccino",
        "description": "A rich espresso-based drink topped with steamed milk and foam.",
        "price": 32,
        "stock_quantity": 10,
        "ingredients": [
            {
                "stock_id": "b6997b2e-1536-47f5-81f6-b607639a0ea7",
                "item": "Coffee Beans",
                "unit_type": "grams",
                "quantity": 1
            },
            {
                "stock_id": "426b7f23-b065-429c-8b67-017f8bec3dc1",
                "item": "Sugar",
                "unit_type": "grams",
                "quantity": 1
            }
        ]
    },
    {
        "id": "5f577d03-b41f-40c3-a56f-e28a37d4b7a0",
        "name": "Iced Coffee",
        "description": "Ice Coffee.",
        "price": 35,
        "stock_quantity": 15,
        "ingredients": [
            {
                "stock_id": "f4742e76-92aa-4c7b-b3c4-6c230337ffbd",
```


2.4 Update product

Purpse: Updates product
Method: PUT
Path: /product

Request Body

- `product` (uid or string)
    - Name of ID of product.
- `updates` (array)
    - Array of fields to be updated, and their values.
- `ingredients` (array)
    - Array of items to be updated.

        - stock_item (uid or string)
            - ID or Name of stock item to be updated.
        - quantity (float)
            - New quantity

If stock_item is not already part of ingredients it gets added.

If sotck_item does not exist its indicated in missingStockItems

If stock_item quantity is set to 0 it gets removed from the ingredients.

If stock_item is already part of ingredients quantity gets updated.

```
{
  "product": "{ID or Name}",
  "updates": {
    "name": "Test Drink Update",
    "price": 20.00
  },
  "ingredients": [
    { "stock_item": "Coffee Beans", "quantity": 0 },
    { "stock_item": "Sugar", "quantity": 1 },
    { "stock_item": "Cream", "quantity": 2 },
    { "stock_item": "NA", "quantity": 2 }
  ],
  "missingStockItems": []
}
```

Response

On a successful request, the server responds with a status code of 200 OK and a JSON object containing the following structure:

- success (boolean):
    - Indicates whether the signup was successful.
- message (string):
    - A message providing additional information.
- product (object):
    - Product in its current state (after update).
- Ingredients (array):
    - Array of ingredients used with product (after update).

2.5 Delete product

Purpose: Deletes product
Method: PUT
PATH: /product/{id}

Request Body

None.

Response

On a successful request, the server responds with a status code of `200 OK` and a JSON object containing the following structure:

- `success` (boolean):
  - Indicates whether the signup was successful.
- `message` (string):
  - A message providing additional information.
- `product` (object):
  - Deleted product.


```
{
    "success": true,
    "message": "Product 05686746-c0db-4644-b90e-c55f0e0f6ffd deleted successfully",
    "product": {
        "id": "05686746-c0db-4644-b90e-c55f0e0f6ffd",
        "name": "Test Drink",
        "description": "A drink that was deleted.",
        "price": 20,
        "stock_quantity": 10
    }
}
```

3. Orders

3.1 Create order

Purpose: Creates a new order
Method: POST

```
{
  "products": [
    {
      "product": "Iced Coffee",
      "quantity": 1,
      "custom": { "note": "Put milk before hot water" }
    },
    {
      "product": "Iced Coffee",
      "quantity": 1,
      "modifications": [
        { "stock_item": "Sugar", "action": "remove", "quantity": 5 },
        { "stock_item": "Milk", "action": "add", "quantity": 10 }
      ]
    },
    {
      "product": "Latte",
      "quantity": 1
    }
  ]
}
```

3.2 Get orders

Purpose: Retrieves all orders
Method: GET
Path: /get_orders

4. Stock

4.1 Create Stock
Purpose: Creates new stock items
Method: POST
Path: /stock

Request Body Parameters:

Items need to be passed in as an array [...]. Add any parameters to be updated along with its updated value.

- item (string): The name of the item to be added.
- quantity (decimal): The quantity of the item being added.
- unit_type (string): The unit type for the quantity.
- max_capacity (decimal): The maximum capacity for the stock item.
- reserved_quantity (decimal): Quantity that is reserved by an order. [optional]


```
[
  {
    "item": "Cream",
    "quantity": 100,
    "unit_type": "kg",
    "max_capacity": 200,
    "reserved_quantity": 10
  },
  {
    "item": "Water",
    "quantity": 50,
    "unit_type": "liters"
  }
]
```


4.2 Get stock

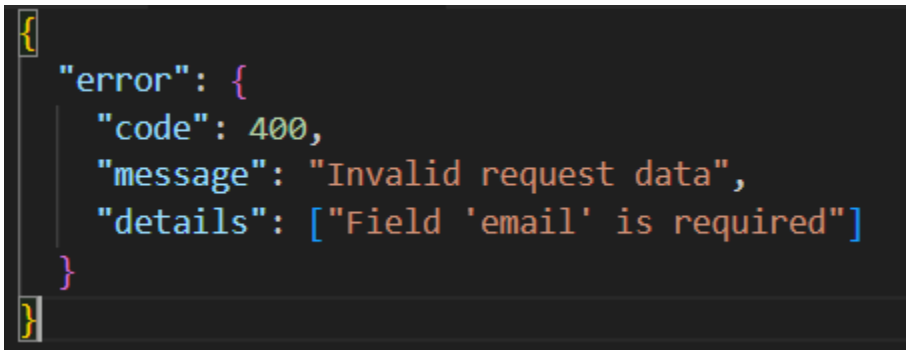
Purpose: Retrieves all stock in the database

Method: GET

Path: /get_stock

Error handling

- Standardized JSON format
- Example of a response is provided below.



```
{
  "error": {
    "code": 400,
    "message": "Invalid request data",
    "details": ["Field 'email' is required"]
  }
}
```

Coding Standards

- Languages: TypeScript, Next.js, React Native, Node.js
- Style: ESLint, Prettier, naming conventions
- Folder/repo structure
- Commit messages: Conventional Commits
- Branching & PR review process
- Testing standards: Jest, React Testing Library

We expand on this in more detail in our Coding Standard document

---

Technical Installation Manual

- Overview of system components
- Prerequisites: Node.js, Docker, Git, etc.

- Installation steps: Clone repo, install dependencies, environment variables
- Running services: Docker Compose, npm start
- Mobile setup: Expo, emulator/phone
- Testing instructions with screenshots

Testing

- Unit testing: order totals, stock updates
- Integration testing: end-to-end service flows
- Frontend UI tests: React Testing Library
- Automated execution: GitHub Actions workflows
- Coverage: 90% on critical modules

CI/CD

- GitHub Actions workflows overview
- CI: linting, unit tests, integration tests
- CD: Docker build & push, deploy
- Rules: PR must pass tests before merge
- Example YAML snippets

Security & Roles

- Authentication: JWT
- Role-based access control: Customer, Barista, Manager, Financial Manager
- Security: HTTPS, CORS, rate limiting, input validation
- Logging: sensitive events (logins, failed attempts)

Deployment Description

Target Environment

The Coffee Shop Management System is deployed in a cloud-based environment, utilising Render to host containerised services and Supabase for database and authentication. This setup ensures scalability, high availability, and reduced infrastructure management overhead.

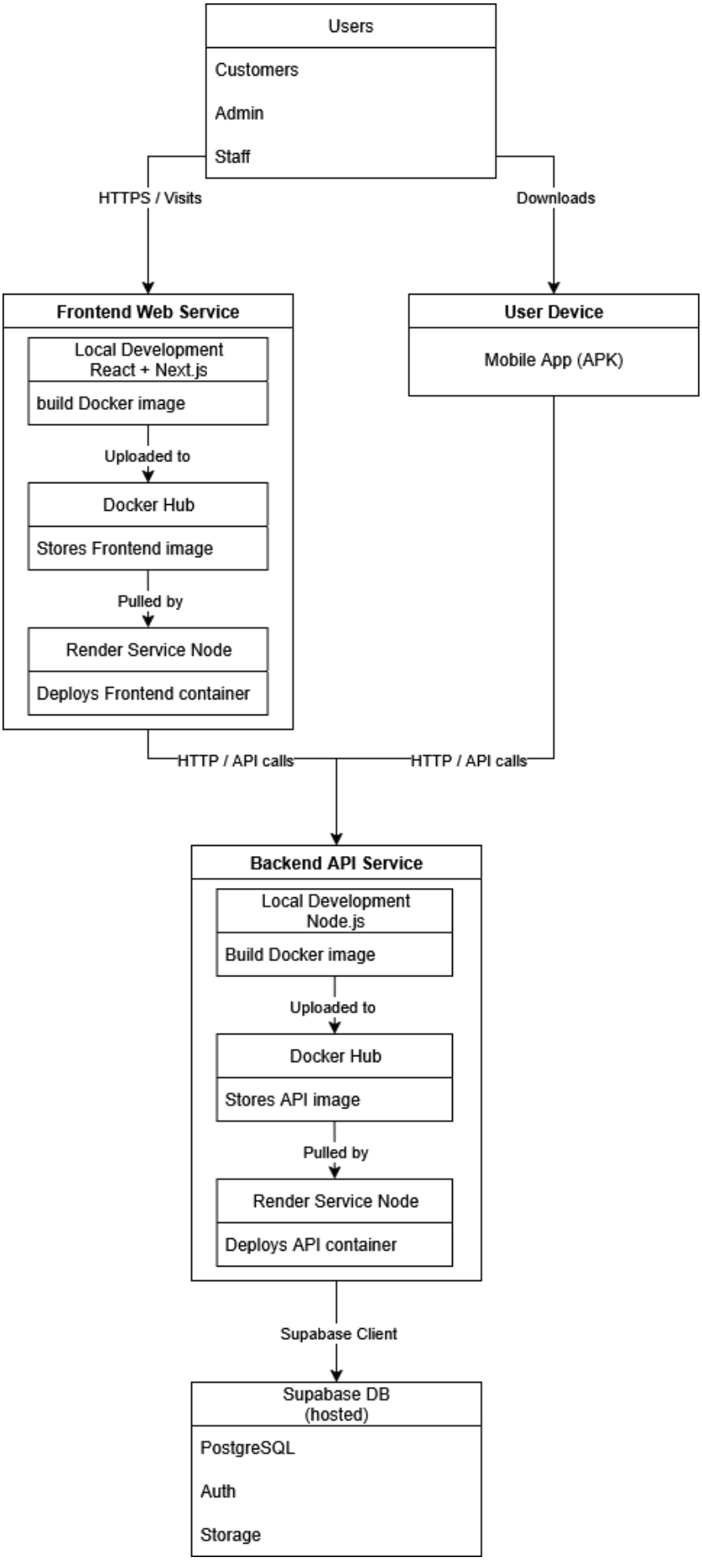
Deployment Topology

The system follows a multi-tier, containerised architecture consisting of the following components:

1. Frontend Web Service
    - Developed with React + Next.js.
    - Packaged as a Docker image and pushed to Docker Hub.
    - Deployed on Render as a containerised web service.
    - Provides the web-based interface for customers, staff, and administrators.
2. Mobile Application (Android)

- Built locally using Android Studio and packaged as an APK.
- Distributed directly to end-user devices.
- Communicates securely with the backend API over HTTPS.

3. Backend API Service
    - Handles core business logic such as ordering, inventory management, and administrative functions.
    - Built and packaged as a Docker image, then pushed to Docker Hub.
    - Deployed on Render as a containerised API service.
    - Exposes RESTful endpoints consumed by both the frontend and mobile app.
4. Database Service
    - Powered by Supabase (managed PostgreSQL).
    - Provides secure, cloud-hosted data storage, authentication, and real-time updates.
    - Accessible only to the backend API service for controlled data flow.


Users
Customers
Admin
Staff
HTTPS / Visits
Downloads
Frontend Web Service
Local Development
React + Next.js
build Docker image
Uploaded to
Docker Hub
Stores Frontend image
Pulled by
Render Service Node
Deploys Frontend container
User Device
Mobile App (APK)
HTTP / API calls
HTTP / API calls
Backend API Service
Local Development
Node.js
Build Docker image
Uploaded to
Docker Hub
Stores API image
Pulled by
Render Service Node
Deploys API container
Supabase Client
Supabase DB
(hosted)
PostgreSQL
Auth
Storage

Tools and Platforms

- Docker – Containerisation of frontend and backend services.
- Docker Hub – Image registry for storing and versioning service containers.
- Render – Hosting and deployment of containerised services.
- Supabase – Managed PostgreSQL database with authentication and storage.
- Android Studio – Development and packaging of the mobile APK.

Quality Requirements Support

- Scalability – Render enables dynamic scaling of web and API containers; Supabase supports horizontal and vertical database scaling.
- Reliability – Containerised deployment ensures service isolation and minimises downtime, while cloud hosting enhances availability.
- Maintainability – Independent deployment of frontend, backend, and mobile clients; Docker streamlines version control and redeployment.

Wow Factors

- Push notifications
- Gamification
- Low-stock alerts

Versioning

- History of SRS updates:
    - v1: Initial
    - v2: Design patterns & constraints
    - v3: Full documentation for Demo 3
