Bakery-management-system

Design Decisions Report

NAME: ISHITA GUPTA
SAP ID: 500105749
ROLL NO.: R2142220088
BATCH: CCVT-B7

Design Decisions Report – Bakery Management System

---

1. Microservices Architecture with Docker

Decision:
The system is composed of multiple containers – PostgreSQL, Redis, Flask Backend, React Frontend, RabbitMQ, and a Worker Service – all orchestrated using Docker Compose.

Reasoning:

- Enables isolated development and deployment of each service.
- Simplifies maintenance and debugging.
- Supports scalability, allowing each service to scale independently based on load.

---

2. PostgreSQL for Relational Data Management

Decision:
PostgreSQL was selected as the primary database for storing bakery products, orders, and order items.

Reasoning:

- Relational structure fits well with normalized schemas (products, orders, order_items).
- Offers strong ACID compliance, ensuring consistency and reliability of transactional data.

---

3. Redis for Caching

Decision:
Redis was implemented to cache product listings and reduce database load.

Reasoning:

- Boosts performance by serving frequent requests from memory.
- Minimizes latency, improving frontend responsiveness.
- Smart cache invalidation ensures data accuracy after stock updates.

---

4. Flask Backend with RESTful APIs

Decision:
The backend is developed using Python Flask, exposing RESTful APIs for the frontend and handling order workflows.

Reasoning:

- Lightweight and fast for small-scale API services.

- Integrates easily with PostgreSQL, Redis, and RabbitMQ.
- Clear separation of logic for product management, orders, and status retrieval.

---

5. React Frontend for UI

Decision:
React.js was chosen for building the user interface, allowing customers to browse, select, and order bakery products.

Reasoning:

- Dynamic and responsive UI for an enhanced user experience.
- Modular components make it easy to extend features like filtering or user login in the future.
- Communicates with backend via Axios, providing real-time updates.

---

6. RabbitMQ for Asynchronous Processing

Decision:
RabbitMQ is used to queue order processing tasks which are then handled by the worker service asynchronously.

Reasoning:

- Prevents the backend from being blocked during long-running tasks.
- Ensures smooth handling of high volume orders by decoupling order placement from processing.
- Reliable with message persistence and retry options.

---

7. Worker Service for Background Order Processing

Decision:
A dedicated Python-based worker service listens to the RabbitMQ queue and processes orders by updating the database.

Reasoning:

- Keeps order processing independent of the main API, increasing throughput.
- Allows future extension (e.g., sending email confirmations or SMS alerts).

---

8. Health Checks and Resource Limits

Decision:
Health checks are implemented for all containers, and resource limits are defined in docker-compose.yml.

Reasoning:

- Ensures system stability by automatically restarting failed services.
- Prevents resource starvation, especially in low-memory environments.

---

9. Security and Best Practices

Decision:
Sensitive configuration values are stored in environment variables using a .env file.

Reasoning:

- Protects sensitive data from being hardcoded in source files.
- Enhances portability across development, staging, and production environments.

---

10. Future-Proofing and Extensibility

Decision:
The system is designed with future enhancements in mind such as authentication, payments, monitoring, and CI/CD.

Reasoning:

- Modular design makes it easy to plug in new features.
- Containerization facilitates automated deployment pipelines and testing.
