Decision Report – Food Delivery Platform

Architecture Overview

- Architecture Style: Modular Monolithic
- Database: SQLite
- Tools & Services:
  - Redis (Azure) for Pub/Sub
  - Azure Web App for deployment
  - Azure Blob Storage for image uploads

- Testing URL: https://gsg-fgcfh9anhaeqaff9.uaenorth-01.azurewebsites.net/

(Open two windows: one as Customer and one as Employee, then just press (customer or employee) button to log in.)

Feature 1 — Customer Account Management (Auth, Profile)

- Pattern Choice: Request/Response (REST)
- Reasoning: CRUD operations fit naturally with stateless HTTP. Simple and secure with JWT.
- Alternatives Considered: WebSocket / SSE – unnecessary overhead for non-real-time CRUD.
- Implementation Details: JWT authentication with refresh tokens.

Feature 2 — Order Tracking

- Pattern Choice: Long Polling
- Reasoning: Works everywhere (even without WebSocket). Simpler than SSE under infra constraints.
- Trade-offs:
  - Short polling: Possible but drains battery and creates many TCP connections.
  - **SSE: not necessary for exact real-time; better to keep connection free for other threads.**
  - **ليس مهم ان يصل التحديث في نفس اللحظة , لذلك عند قطع الاتصال مع ال client يستطيع اخر ان يأخذ مكانه اذا كان هناك ضغط على السيرفر و يذهب ال polling connection/thread الى ال connection , اما ال sse يبقى حاجز ال incoming queue . thread**
- Implementation Details:
  - Endpoint with 60-second timeout.
  - Client sends the last known status; if it equals the DB status, it waits until change or timeout.
  - Polling stops when status = Delivered or Cancelled.

Feature 3 — Driver Location

- Pattern Choice: Server-Sent Events (SSE)
- Reasoning: One-way streaming is suitable for continuous driver → client updates. Lightweight compared to WebSockets.
- Trade-offs: One-way only; no need for WebSocket.
- Alternatives Considered: Long polling – most research recommends SSE when supported by client.
- معظم المفالات تفضل استخدام ال sse في حال ان ال client يدعمه و فقط التوجه لل long polling اذا لم يكن ال client يدعم ال sse
- Implementation Details:
  - SSE headers, one stream per driver.
  - For testing: JavaScript updates location every 45 sec → SSE sends to client when changed.

Feature 4 — Restaurant Order Notifications

- Pattern Choice: SSE + Redis Pub/Sub
- Reasoning: Real-time fan-out to multiple employees. Redis ensures scalability across instances.
- Trade-offs: One-way only; Redis infrastructure required.
- Alternatives Considered:
  - Pure SSE: not scalable across servers; connections would consume memory(queue of clients) and all client connection with server , but here all client connection with redis .(assume there is many employees)
  - WebSocket: unnecessary complexity for one-way notifications.
- Implementation Details: Backend subscribes to Redis channel and streams events to SSE clients.
- Additions: Uses managed Redis service (use redis from azure).

Feature 5 — Customer Support Chat

- Pattern Choice: WebSocket
- Reasoning: Bi-directional, low-latency communication is essential for chat.
- Alternatives Considered: SSE – one-way, not suitable for chat.
- Implementation Details:
  - Socket.IO with per-session rooms (each chat has an ID).
  - Messages persisted in DB.
  - Chat only between customers and employees.

Feature 6 — System-Wide Announcements

- Pattern Choice: SSE + Redis Pub/Sub
- Reasoning: Efficient broadcast to thousands of users. Works with clustered deployments.
- Trade-offs: One-way only; acknowledgements require REST. Redis is required.

- Alternatives Considered:
  - Pure SSE: **not scalable across servers; connections would consume memory(queue of clients) and all client connection with server , but here all client connection with redis .**
- - WebSocket: unnecessary its bidirectional and waste resources.
- Implementation Details:
  - REST endpoint for creating announcements → stored in DB → published to Redis.
  - Clients fetch history on load + listen for live SSE.

Feature 7 — Image Upload Processing

- Pattern Choice: Short Polling
- Reasoning: Simple periodic status checks are enough for batch-style jobs. Avoids long-lived connections.
- Alternatives Considered: SSE/WebSockets – real-time but overkill for async uploads.
- Implementation Details:
  - Job table stores upload ID + status.
  - POST to create job, GET for status polling.
  - Images uploaded to Azure Blob, processed asynchronously (e.g., background removal), then returned to user.

Challenges and Difficulties

The biggest issue during testing was that the app only allowed one browser tab to connect at a time. When opening more than one tab, it froze and produced a timeout. This problem took around 3 days to debug. After reviewing Flask's threading behavior (when discuss it in the Wednesday lecture), it became clear that Flask is single-threaded and supports only one active connection. However, the system required multiple concurrent connections (Long Polling, SSE, WebSocket). The solution was to switch to a more suitable server stack that supports concurrency:
- Eventlet
- Gunicorn
- Gevent
- Gevent-WebSocket

This setup provides event-loop-like behavior similar to JavaScript, preventing Flask from being blocked by multiple connections.

/**/

You can test features 1-6 via :

https://gsg-fgcfh9anhaeqaff9.uaenorth-01.azurewebsites.net/