Backend Communication Patterns Project: Food Delivery Platform

feature1_account_management :

Pattern Chosen:

Request/Response (Synchronous with Authentication & Transactions)

Business requirement analysis:

Manage user accounts securely , best for async and immediate feedback

Technical considerations:

sync req/res, PostgreSQL transactions, JWT for stateless auth, DB connection retries

User experience impact:

Quick feedback, secure access, clear error messages.

Scalability factors:

JWT is stateless easy to scale, retries for DB improve reliability.

Alternatives considered:

pub/sub: rejected not usable when the user expects immediate confirmation like login.

WebSocket's : rejected overkill for account management.

Trade-offs accepted:

Simple sync design but less scalable under very high load; JWT harder to revoke.

feature2_order_tracking:

Pattern Chosen:

Request/Response (for order CRUD) + Server-Sent Events (for live status)

Business requirement analysis:

Customers must see real-time updates

Technical considerations:

sync requests for CRUD, SSE for live updates, PostgreSQL transactions

User experience impact:

Instant order status updates, clear info

Scalability factors:

Works for small/medium traffic; needs Redis for many users.

Alternatives considered:

WebSockets : rejected for one-way updates

Polling : wasteful, higher latency

Trade-offs accepted:

SSE is simple but only one-way; doesn’t scale well without Redis or message broker, and limited under high load

feature3_driver_location :

Pattern Chosen:

WebSocket + REST fallback

Business requirement analysis:

Track driver location in real-time during delivery

Technical considerations:

WebSocket for live updates, Redis + in-memory storage for caching, PostgreSQL for order validation.

User experience impact:

Customers see driver location live, updates are accurate and frequent.

Scalability factors:

Works for moderate traffic; Redis helps scale.

Alternatives considered:

Polling : slow/inefficient, can increase server load with many open connection

SSE : not ideal for bidirectional updates.

Trade-offs accepted:

WebSocket adds complexity and slightly higher battery usage but gives real time tracking , REST fallback ensures reliability.

feature4_restaurant_notifications :

Pattern Chosen:

SSE (Server-Sent Events) + REST fallback

Business requirement analysis:

Notify restaurants in real-time of new orders

Technical considerations:

SSE for live order notifications, Redis Pub/Sub for broadcasting events, PostgreSQL to store orders and restaurant data.

User experience impact:

Restaurants receive instant order notifications without polling.

Scalability factors:

Redis Pub/Sub handles multiple restaurants efficiently , and dont miss order

Alternatives considered:

WebSockets: bidirectional, unnecessary for simple notifications and slightly higher complexity and overhead than SSE

polling: inefficient, higher latency than WebSocket's/SSE

Trade-offs accepted:

SSE only allows server client updates, but sufficient for order notifications

feature5_support_chat :

Pattern Chosen:

WebSocket (Socket.IO) + REST fallback

Business requirement analysis:

Enable real-time two-way chat between users and support agents.

Technical considerations:

WebSockets for low-latency bidirectional messaging, PostgreSQL to store chat messages and user info, REST endpoints for retrieving chat history.

User experience impact:

Users can send/receive messages instantly, see typing indicators, and get delivery confirmations.

Scalability factors:

WebSockets handle multiple simultaneous chat rooms efficiently, PostgreSQL ensures message persistence.

Alternatives considered:

SSE: insufficient for bidirectional chat, Polling: inefficient, wastes resources, increases latency

Trade-offs accepted:

Requires persistent connections, WebSocket overhead is acceptable for interactive support chats.

feature6_announcements :

Pattern Chosen:

REST API + Server-Sent Events (SSE) + Redis Pub/Sub

Business requirement analysis:

Deliver system-wide or targeted announcements to users in real-time and persist them for later retrieval.

Technical considerations:

REST endpoints for creating, retrieving, and tracking announcements, **SSE** for real-time updates pushed to clients, **Redis Pub/Sub** for broadcasting announcements efficiently, PostgreSQL to store announcements and user-read statuses.

User experience impact:

Users receive instant notifications for critical updates, promotions, or maintenance messages, while keeping a history for offline access.

Scalability factors:

SSE and Redis allow thousands of concurrent users to receive real-time announcements, database ensures persistence.

Alternatives considered:

WebSockets overkill if few minutes delay is acceptable, Short polling too much server load (all users polling).

Trade-offs accepted: SSE is unidirectional, browser support is sufficient. Persistent connections may slightly increase server load, mitigated via Redis Pub/Sub.

feature7_image_upload :

Pattern Chosen:

REST API + SSE + Celery + Redis Pub/Sub

Business requirement analysis:

Allow restaurants to upload images (menu items, logos) with asynchronous processing and real-time status updates.

Technical considerations:

REST endpoints for image upload, retrieval, and job status, **SSE** for real-time updates on upload and processing progress, **Celery** for background processing of images (resizing, compression, thumbnail generation), **Redis Pub/Sub** for broadcasting processing status, **PostgreSQL** to track upload jobs and store metadata.

User experience impact:

Users (restaurant admins) can upload images and track progress live, ensuring transparency and reliability.

Scalability factors:

Celery with Redis allows concurrent image processing jobs without blocking the main API. SSE ensures many clients can track progress without polling.

Alternatives considered:

Could use WebSockets, but SSE fits unidirectional progress updates well. Direct synchronous processing was rejected due to performance and blocking concerns.

Trade-offs accepted:

SSE streams remain open for long periods, cleanup and retries must be managed carefully. Image processing time depends on file size, mitigated with Celery.