# Backend Communication Patterns: Food Delivery

#### By: Zain Refqi Assaf

## Feature 1: Customer Account Management

### Login:

I will use the sync req-res pattern.

**Business requirements:** The customer must be able to securely log in before accessing any features in the app. The authentication process must be instant, ensuring the user is who they claim to be.

**Technical Considerations:** A synchronous HTTP POST request is used. The server verifies the credentials. If successful, a response containing a JWT (JSON Web Token) is returned to the client. The client uses this token in all future requests to verify its identity.

**User experience:** The user feels the process is fast and smooth. They receive immediate feedback on success or failure, eliminating the need to wait.

**Scalability factors**: The Req-Res pattern relies on stateless connections, meaning the server doesn't need to maintain the client's state between requests. This allows the server to handle a very large number of concurrent login requests with great efficiency.

**Alternatives considered:** Long Polling or WebSocket: These patterns are rejected because they are designed for real-time updates and are not necessary for a one-time login.

**Trade-offs accepted: nothing**

### Profile Updates

I will use the sync req-res pattern.

**Business requirements:** Customers should be able to modify their personal information (such as name, email, password). The user should receive immediate confirmation of a successful change.

**Technical Considerations:** A synchronous HTTP PUT request is used. The server validates the JWT token, updates the data in the database, and then returns a response indicating success or failure.

**User experience:** The user expects changes to be immediate. They are immediately notified of a successful update or any potential errors.

**Scalability factors**: This process is also stateless and easily scalable. Each request is handled independently without affecting other requests.

**Alternatives considered:** nothing

**Trade-offs accepted:** nothing

### Payment

The synchronous Req-Res pattern is used for the core process, with asynchronous processes used for secondary tasks in the background.

**Business requirements:** Card, balance, and security verification must be immediate. The customer must receive immediate confirmation of the success or failure of the payment. Secondary processes (such as sending an email receipt) can be delayed and run in the background.

**Technical Considerations:** A synchronous HTTP POST request is sent. The server performs card validation and authorization with the payment gateway. Once the payment is approved, the server sends an immediate success response to the customer. At the same time, other asynchronous processes are initiated in the background, such as sending an email notification to the customer or sending a report to a fraud detection system.

**User experience:** The user receives immediate confirmation that the payment was successful, which makes them feel secure and allows them to move on to the next step immediately. The user does not have to wait for secondary processes to complete.

**Scalability factors**: By separating primary processes from secondary processes, the main server is quickly freed up to process other payment requests, increasing the system's efficiency and its ability to handle a large number of simultaneous processes.

**Alternatives considered:** WebSocket update model is not appropriate because the client requires an immediate and final result, not just incremental updates.

**Trade-offs accepted:** A slight delay in non-critical processes (such as email receipts) is acceptable in exchange for high speed and responsiveness for the primary process (payment confirmation).

## Feature 2: Order tracking for customers (checking the status)

I will use the short polling, because the process as follows: the client sends the request and the server response with job-id to allow the client checking the status every 30 sec or 2 mins, technically, it is a req-res but broken into steps but with job-id it becomes polling.

**Business requirements:** The clients need to know about their order status in near real time.

**Technical Considerations:** Minimize the number of requests, and any client can work with http (easy to implement)

**User experience:** the user did not notice the delay because it is a near real time.

**Scalability factors**: if it was +1000 online on the same time, the system can handle them because the requests are distributed in interval time and there are no open connections for long time that’s mean less resources.

**Alternatives considered:** WebSocket but it is rejected because real-time updates are not strictly required, and it needs persistent connections that’s mean drains battery, long polling also rejected because it needs many open connections.

**Trade-offs accepted:** long polling: I chose the probability of wasting some requests when nothing change but it is okay because my goal is simplicity, WebSocket: I know there is some delay, but I need to maintain the battery life and minimize hits on server.

## Feature 3: Driver Location Updates

I will use the SSE pattern

**Business requirements:** this pattern will be updating the user in real time which means they can see the driver’s location in 10-15 seconds, and view it on the map without needing to refresh.

**Technical Considerations:** It will utilizelong lived http open connections.

**User experience:** the user can see the location smoothly and it is real-time -10/15 seconds-, if the connection on mobile is lost the app will try to open SSE connection auto.

**Scalability factors**: it is built on http, so it is easy to work with proxies and load balancers.

**Alternatives considered:** Long polling: it causes an overhead because it needs every time open and close connections for every update. WebSocket: it is more complex and I don’t need two way directions.

**Trade-offs accepted:** In this case, I have not made any trade-offs because SSE is the best option from all perspectives.

## Feature 4: Restaurant order notifications

I will use the SSE and the pub-sub patterns

**Business requirements:** the restaurant dashboardshould be updated all the time without needing to refresh for all devices on the restaurant and in real time.

**Technical Considerations:** when the customer place an order, the server - the publisher -will be publish the event on the topic and broadcast it to all subscribers -notification service- then the notification service opens a SSE connection to all dashboards.

**User experience:** it is a good experience, the orders will be show instantly on dashboard so there is no delay, and all the devices will be sync together.

**Scalability factors**: Pub/Sub ensures efficient event distribution even if you have thousands of restaurants. Lightweight SSE (one-way connection).

**Alternatives considered:** WebSocket: Provides bidirectional connectivity but is more complex and consumes more resources. It's not necessary because our communication is one-way (server → dashboard). Long polling: Technically easier, but very expensive (many empty requests) and impractical for thousands of restaurants.

**Trade-offs accepted:** We accept the complexity, in exchange for real-time, scalability, and a better restaurant experience.

## Feature 5: Customer Support Chat

I will use the WebSocket

**Business requirements:** Messages should be in real-time between the customer and agent, handle the number of conversations at the same time and should be persistent, typing indicator and delivery confirmation.

**Technical Considerations:** it will be 2-way connection work over TCP.

**User experience:** The chat interface should be easy to use and feel seamless, Instant feedback, and The ability to scroll through and review past conversations is a key part of the user experience.

**Scalability factors**: the WebSocket connection means that the server does not need to open and close a new connection for each message, significantly reducing resource consumption. And If the number of chats grows significantly, I can be managed by a Vertical and Horizontal Scaling.

**Alternatives considered:** Long Polling: A more refined version where the connection stays open until a message is received, but less efficient than WebSocket because it requires a new connection for every message.

**Trade-offs accepted:** setting up a WebSocket server is more complex than a standard HTTP-based system and higher resource consumption.

## Feature 6: System-Wide Announcements

will use the Pub/Sub model and the SSE protocol.

**Business requirements:** This feature requires the platform to send announcements to thousands of users simultaneously about service outages, new features, or promotions. The announcements are not critical, so they can be delayed for a few minutes. They should not overwhelm the server during peak times.

**Technical Considerations:** The Pub/Sub model is used to meet the requirements of mass broadcasting. The server acts as a publisher, sending the announcement once to a broker or Pub/Sub service. The client application acts as a subscriber, opening a persistent one-way connection with the server using the SSE protocol and waiting for events (announcements) to arrive.

**User experience:** The user receives announcements seamlessly within the application as soon as they are posted by the administrator. The announcements appear automatically on the screen without the need to refresh the page or press any buttons. The user feels like they are always up to date with the latest news from the platform.

**Scalability factors**: The Pub/Sub model separates the server from the client, freeing the server from the burden of sending individual messages to each user. SSE uses a single, persistent connection per user, which is more efficient than opening and closing multiple connections, preventing server overload.

**Alternatives considered:** Short Polling: Rejected because it is inefficient. It requires thousands of users to send repetitive and unnecessary requests to the server to query rare ads, wasting resources. Long Polling: Rejected because it is less efficient than SSE. Although it waits for data, it closes the connection after each message, causing server overload due to the constant re-opening of connections.

**Trade-offs accepted:** The use of a persistent connection on the client side, which consumes some server resources and phone battery, is accepted in exchange for efficient and reliable ad broadcasting.

## Feature 7: Image Upload for Menu Items

I will use the Long Polling pattern

**Business requirements:** the system should upload progress and notify when complete it.

**Technical Considerations:** I used the Multer library to handle upload files over the web, then a long polling to keep the connection open with the client until the processing was complete.

**User experience:** direct note about the progress so keep the user updated all the time and then notify him.

**Scalability factors**: first: this system can handle thousand requests at the same time without any issues because the server receive the image and send immediate response with Job id then start the processing async, so that’s allow server to start serving another request, second: the long polling sit and wait till finish the request, this reduce number of request every seconds so there is no consume the network or CPU resources with increase the users.

**Alternatives considered:** short polling rejected because it consumes a lot of server resources, because it sends very frequent requests (e.g. every second) even if the task state does not change.

**Trade-offs accepted:** Complexity, Server Resources (open connections).