E-Commerce Flash Sale System — Design Document

Advanced Back-end Training-Gaza Sky Geeks

Written By:

-Bayan Aboalrob

-Duaa Braik

Contents:

1. Introduction

E-commerce flash sale systems are designed to handle massive user demand within a very short timeframe—such as during Black Friday or limited product releases. These systems often experience traffic surges of over 100,000 users competing for only a few thousand items.
The goal is to provide fair access, prevent overselling, and maintain performance under extreme load while keeping the user experience smooth and reliable.

2. Common Problems in Flash Sale Systems

Flash sale systems face several core challenges:
- Overselling: Multiple users attempt to purchase the same item simultaneously.
- Inventory Inconsistency: Delayed or incorrect synchronization between cache and database.
- System Overload: Sudden high concurrency can overwhelm backend services.
- Payment Delays: Payment confirmation latency can cause bottlenecks.
- Data Integrity Issues: Distributed transactions increase the complexity of maintaining atomicity and consistency.

3. Our Scenario

In this project, our mission was to design a scalable backend system capable of handling real-world flash sale workloads using two different architectural approaches:
Synchronous HTTP-based and **Asynchronous Event-driven**.
Both were built as microservices, leveraging Nginx as an API Gateway, Redis for high-speed caching, RabbitMQ for messaging, and FlashSaleDB as the system of record.

Sequence and architecture diagrams for both approaches are available in the Google Drive folder for visual reference:

https://drive.google.com/drive/folders/1-Yw2ympF5NFUxOTlfOth1yZaZfjEOJiK

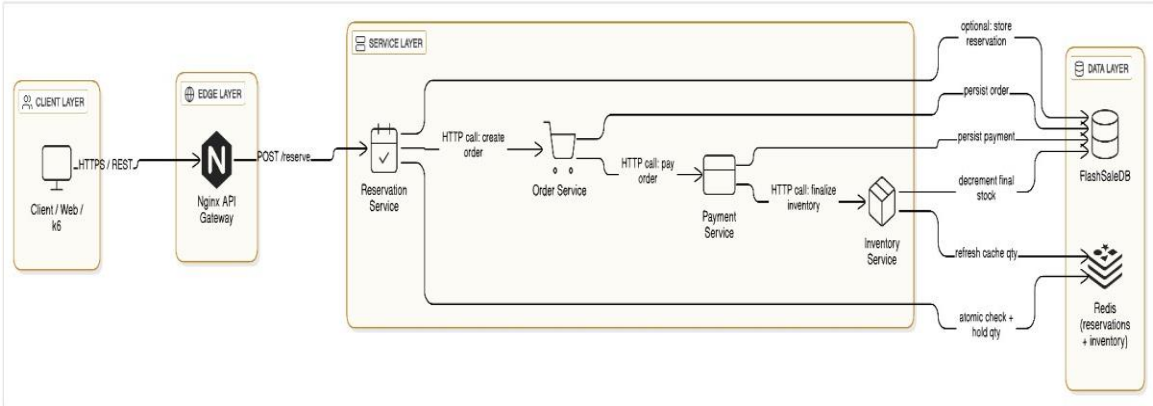
4. System Architecture

4.1 Why Microservices?

We adopted a microservices-based architecture to isolate each business capability — User management, Reservation, Order, Payment, and Inventory — into independently deployable services.
This approach improves scalability, fault tolerance, and maintainability. Each service communicates via HTTP (in synchronous mode) or RabbitMQ events (in asynchronous mode), depending on the implementation.

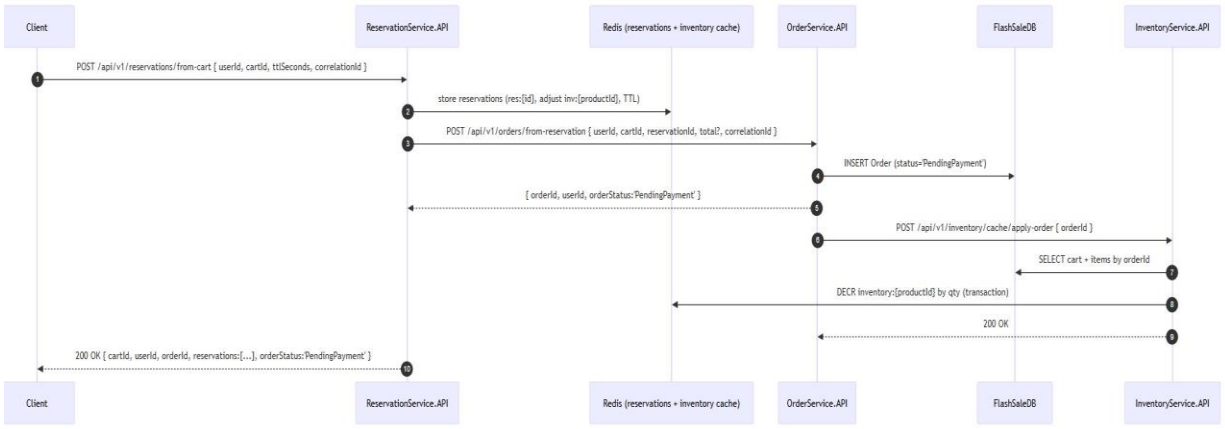
4.2 Synchronous Architecture

In the synchronous model, communication occurs via direct RESTful HTTP calls between services. This ensures strong consistency since each service waits for the next operation to complete before proceeding.

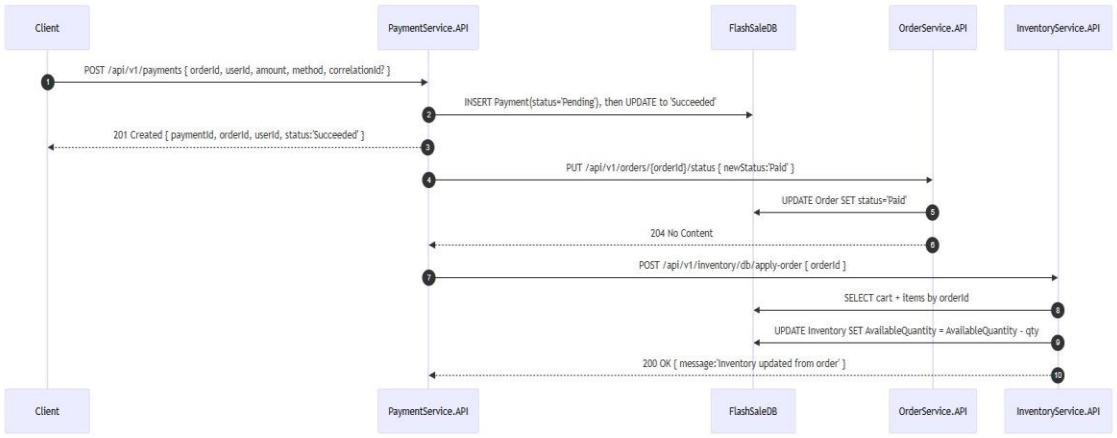


Synchronous Approach Architecture Diagram

Flow Explanation:
In the synchronous architecture, the client initiates a reservation request that passes through Nginx, which routes it to the Reservation Service. The Reservation Service performs an atomic operation in Redis to reserve the requested stock, assigning each reservation a specific time-to-live (TTL) to prevent prolonged stock locking. Once the reservation is successfully stored, it directly calls the Order Service via an HTTP request to create a new order in the PendingPayment state. Afterward, when the user proceeds to payment, the Payment Service processes the transaction, updates the order status to Paid through another HTTP call to the Order Service, and then triggers the Inventory Service to apply the final stock updates in FlashSaleDB. This sequential request–response chain ensures strong consistency and immediate synchronization across all components.



Sequence Diagram for the Synchronous Approach Part (1)



Sequence Diagram for the Synchronous Approach Part (2)

Component Highlights:

- Nginx API Gateway:

Acts as the system's single entry point. It routes external client requests to the correct internal service while handling load balancing, rate limiting, and traffic management.

Main Benefit: Ensures scalability and isolates internal services from direct exposure, enhancing both performance and security.

-Redis Cache (Reservations + Inventory):

Acts as a high-speed, in-memory data store that temporarily holds inventory and reservation data during flash sale operations. It ensures atomic operations using DECR or Lua scripts, preventing race conditions and overselling when thousands of users attempt simultaneous purchases.

Main Benefit: Provides microsecond-level access and atomic updates, drastically reducing database load while enabling real-time consistency and stability under extreme concurrency.

-Reservation Service:

Handles all reservation logic. It stores reservation data in Redis with a defined TTL (time-to-live) to automatically release stock if payment is not completed.

In the synchronous approach, it directly calls the Order Service to create an order in PendingPayment state.

Main Benefit: Acts as the first layer of defense against overselling, ensuring atomic stock reservation and system stability under concurrency.

-Order Service:

Responsible for managing the order lifecycle — from creation to payment confirmation. It stores order details in FlashSaleDB and coordinates with Payment and Inventory services.

Main Benefit: Maintains transaction integrity and order consistency, ensuring that every reserved item corresponds to a valid order.

-Payment Service:

Manages payment authorization and confirmation. Once a payment succeeds, it sequentially triggers updates to both the Order Service (to mark as Paid) and the Inventory Service (to update available quantities).

Main Benefit: Provides a controlled and verifiable payment flow, ensuring reliability and data accuracy across services.

-Inventory Service:

Finalizes stock updates in FlashSaleDB after successful payment and synchronizes Redis cache for real-time accuracy.

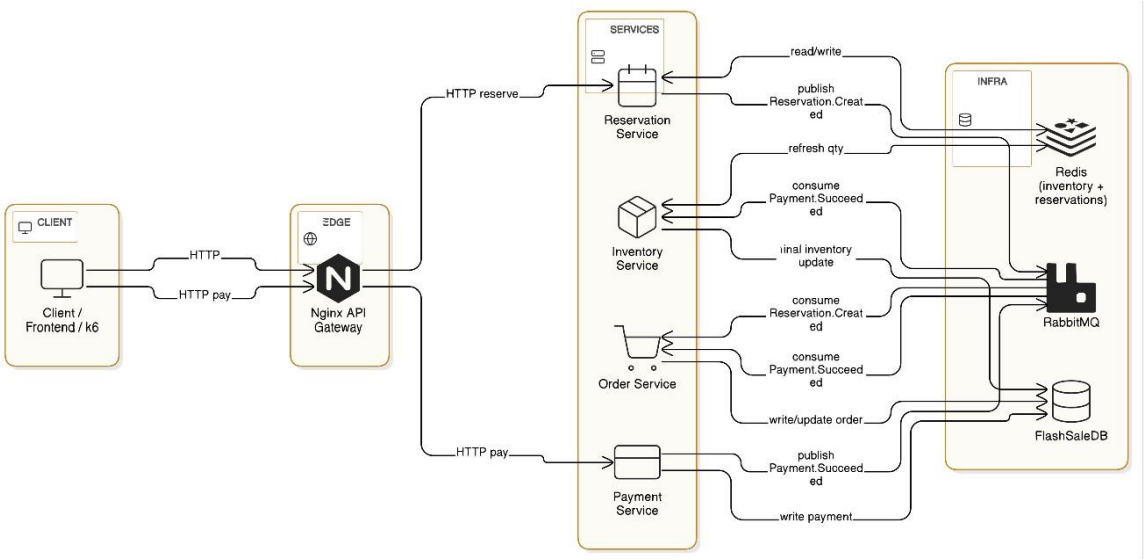
Main Benefit: Guarantees strong data consistency between cache and database, preventing stock mismatches or stale data during flash sale bursts.

-FlashSaleDB (Database):

Acts as the source of truth for all transactions, orders, and payments. It guarantees long-term consistency and data durability once operations are confirmed.

4.3 Asynchronous Architecture (Event-Driven)

In the asynchronous architecture, services communicate using RabbitMQ, decoupling the system and allowing independent scaling of components. Each service emits and consumes events without waiting for direct responses, improving throughput and resilience.

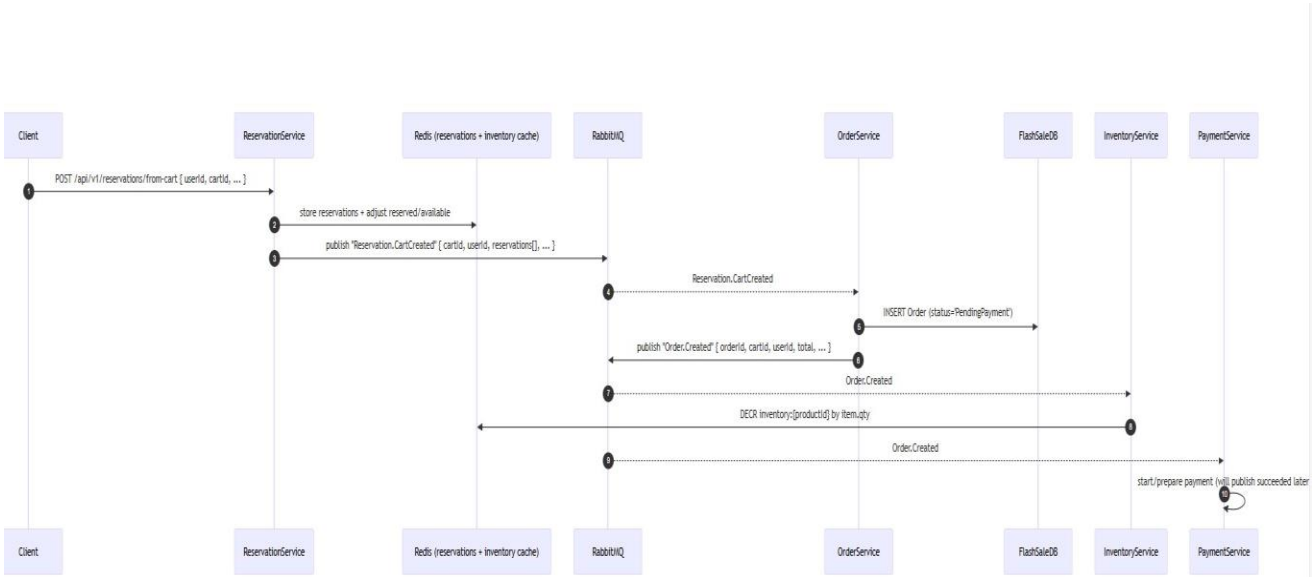


Asynchronous Event-driven Architecture Diagram

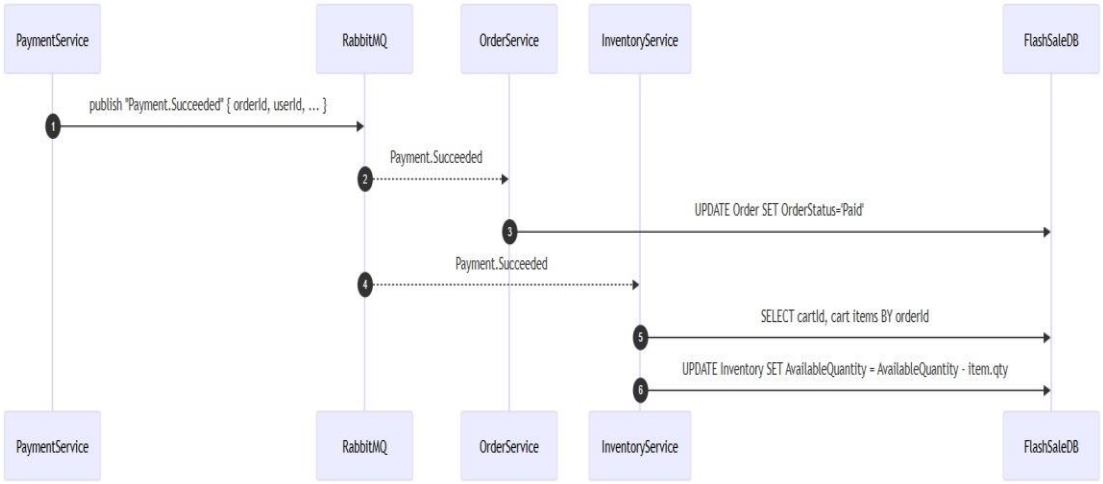
Flow Explanation:
In the asynchronous event-driven architecture, the system's workflow begins when the Reservation Service receives a client request and performs an atomic update to the Redis cache to reserve stock for the requested items. Once the reservation is successfully stored with a defined time-to-live (TTL), the service publishes a Reservation.CartCreated event to RabbitMQ, signaling that a new reservation has been made. The Order Service listens to this event, consumes it, and creates a corresponding order in the database with an initial status of PendingPayment.

After successfully creating the order, it emits an Order.Created event, which can trigger downstream actions such as logging or analytics updates. When a user proceeds with payment, the Payment Service processes the transaction and, upon successful confirmation, publishes a Payment.Succeeded event.

This event is then consumed by both the Order Service and the Inventory Service — the former updates the order's status from PendingPayment to Paid, while the latter finalizes inventory adjustments in the database and synchronizes the Redis cache to reflect the new stock levels. Through this asynchronous flow, each microservice operates independently, ensuring high scalability, non-blocking communication, and reliable event-driven coordination across the system.



Sequence Diagram for the async approach part(1)



Sequence Diagram for the async approach part(2)

Component Highlights:

-Nginx API Gateway:

Acts as the system's unified entry point, managing all external traffic and routing requests to internal microservices. It provides load balancing and shields backend services from direct client exposure.

Main Benefit: Ensures secure and scalable traffic distribution, optimizing performance under heavy load and maintaining isolation between clients and backend components.

-RabbitMQ (Message Broker):

Serves as the communication backbone for asynchronous event exchange between services. It guarantees reliable message delivery through acknowledgments, retries, and queue durability, buffering spikes in demand during flash sales.

Main Benefit: Enables loose coupling and horizontal scalability, allowing services to function and recover independently while maintaining system continuity even under extreme concurrency.

-User Management Service:

Handles user registration, authentication, and token management for both clients and internal microservices. It issues JWT or API tokens during login and supports token refresh workflows for session longevity. This service also integrates with Nginx for route protection and access control.

-Reservation Service:

It performs atomic stock reservations in Redis, assigning each reservation a TTL (time-to-live) to prevent indefinite holds on inventory. Once confirmed, it publishes a Reservation.CartCreated event to RabbitMQ to trigger order creation.

Main Benefit: Delivers high-speed, non-blocking reservation handling with atomic guarantees, preventing overselling and maintaining accuracy under massive concurrent user activity.

-Order Service:

Consumes Reservation.CartCreated events, creates new orders in FlashSaleDB, and emits Order.Created events for tracking and downstream synchronization. It also listens for Payment.Succeeded events to update order statuses to Paid.

Main Benefit: Provides event-driven order lifecycle control, ensuring consistent state transitions and reliable order processing without tight coupling between services.

-Payment Service:

Processes payment authorizations and confirmations asynchronously. Upon successful completion, it publishes a Payment.Succeeded event to RabbitMQ, prompting updates in the Order and Inventory services.

Main Benefit: Offers resilient and modular transaction processing, isolating payment logic while guaranteeing dependable communication with other services.

-Inventory Service:

Subscribes to Payment.Succeeded events, finalizing inventory updates in FlashSaleDB and synchronizing the Redis cache to reflect the latest stock availability.

Main Benefit: Ensures eventual consistency between cache and database, maintaining accurate real-time inventory data and preventing discrepancies during high-volume transactions.

-Redis Cache (Reservations + Inventory):

Serves as an in-memory storage layer for reservations and inventory data, enabling atomic operations like DECR or Lua scripts to prevent overselling. It maintains real-time inventory snapshots to support rapid reads and writes during flash sale peaks.

Main Benefit: Provides microsecond-level access and concurrency safety, reducing database load and ensuring instantaneous stock synchronization across services.

-FlashSaleDB (Database):

Acts as the source of truth for all transactions, orders, and payments. It guarantees long-term consistency and data durability once operations are confirmed.

5. Handling Overselling and Concurrency

During a flash sale, thousands of users may try to buy the same item at once. To keep things fair and prevent the system from crashing or overselling, we've added several smart mechanisms that work together to manage concurrency and keep data consistent:

-Atomic Operations in Redis:

Every reservation or stock update happens as a single, atomic action using Redis DECR commands or Lua scripts. This means each operation is processed one at a time, guaranteeing that no two people can reserve the same item simultaneously — even under massive traffic.

-Reservation TTLs (Time-to-Live):

Each reservation stored in Redis has an expiration timer (TTL). If a customer doesn't complete their payment within that time, the reservation automatically expires, and the item goes back into available stock. This keeps the system clean and prevents stock from being locked by abandoned carts.

-Idempotency Across Services:

All services — from Reservation to Order, Payment, and Inventory — are designed to handle duplicate requests safely. Even if the same event is received twice due to network retries, the system processes it only once. This avoids double payments or duplicate orders and keeps data accurate.

-Horizontal Scaling and Load Distribution:

To handle high demand, each service runs multiple instances that can process requests in parallel. RabbitMQ helps spread the workload evenly across these instances, ensuring no single service becomes a bottleneck. This makes the system fast, reliable, and ready to scale during heavy flash sale traffic.

6. Comparative Analysis

Here is a concise comparison between the synchronous and asynchronous architectures, highlighting their key characteristics, trade-offs, and performance behaviors under flash-sale conditions.

Field	Synchronous	Asynchronous
Communication	Direct HTTP	Event Bus (RabbitMQ)
Consistency	Strong (Immediate)	Eventual
Fault Tolerance	Low	High
Scalability	Moderate	Horizontal & Independent
Latency	Sequential	Parallel / Queued
Complexity	Low	Moderate–High

7. Patterns Used

Here is the list of patterns used:

-API Gateway Pattern:

Uses Nginx as a single entry point to route, balance, and secure all client requests.

Purpose: Centralize access control and simplify client communication.

Benefit: Improves security, scalability, and isolation of backend services.

-Cache-Aside Pattern:

Employs Redis to cache frequently accessed data (inventory, reservations) before hitting the database.

Purpose: Reduce DB load and boost response speed.

Benefit: Ensures low latency and high throughput under heavy concurrency.

-Saga Pattern (Choreography):

Each service reacts to events (e.g., Reservation → Order → Payment → Inventory) without a central coordinator.

Purpose: Coordinate distributed workflows asynchronously.

Benefit: Achieves eventual consistency with high resilience and no blocking.

-Idempotent Consumers:

Services safely handle duplicate events to prevent inconsistencies.

Purpose: Maintain data integrity in event-driven flows.

Benefit: Ensures reliability and fault tolerance during message retries.

-Write-Behind Strategy:

Database updates occur only after confirmed payments; earlier steps rely on Redis.

Purpose: Defer writes for performance optimization.

Benefit: Balances speed with consistency for flash sale operations.

-Event-Driven Communication:

Using RabbitMQ for asynchronous messaging between services.

Purpose: Decouple and scale microservices independently.

Benefit: Provides elastic scalability and fault isolation across the system.

8. Trade-offs and Design Choices

- Both architectures aim to deliver a fast and reliable flash sale experience but differ in how they balance speed and consistency:

-Synchronous Approach:

Ensures instant consistency between services but is more fragile — if one service fails, the whole process can be affected.

-Asynchronous Approach:

Offers better scalability and fault tolerance since services work independently, though it introduces a short delay before all data becomes consistent.

-Redis:

Plays a key role in both approaches by handling atomic operations and keeping performance high under heavy traffic.