System Design: Scalable Event Ticketing

A high-availability, high-concurrency solution for flash sales with extremely high, spiky traffic

The Challenge: Flash Sales

Design a system to sell tickets for popular events with extremely high, spiky traffic. The system must ensure low latency and prevent oversells while handling massive concurrent demand.

Key challenges include managing 10,000+ reservations per second, maintaining sub-2-second checkout latency, and guaranteeing no double-booking of seats.

Stakeholder Analysis

Buyers

Need fair, fast, and secure access. Real-time seat availability is critical.

Event Organizers

Need platform stability, real-time sales data, and maximized revenue.

Payments/Finance

Need accurate, auditable, and idempotent transactions to prevent errors.

Support

Need to view user orders, resolve issues, and process refunds quickly.

Functional vs. Non-Functional Requirements

Functional (What it does)

- Browse and search for events
- View interactive seat maps
- Select seats for time-limited 10-minute reservation
- Securely process payments
- Receive ticket confirmation

Non-Functional (How it performs)

- **Performance:** p99 checkout latency < 2 seconds
- **Availability:** 99.95% uptime (No SPoF)
- **Scalability:** Handle 10,000+ reservations/sec
- **Maintainability:** Modular microservices

Constraints & Assumptions

1

3rd-Party Payment Gateway

Constraint: Must use external payment gateway like Stripe for transaction processing.

2

Bot Protection Required

Constraint: System must be designed for malicious bots, requiring rate limiting.

3

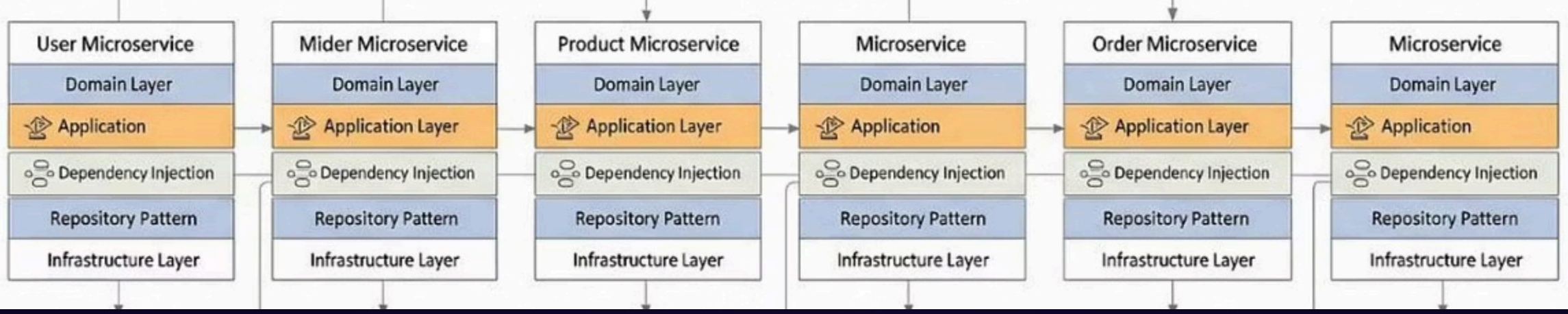
Eventual Consistency

Assumption: Acceptable for event listings and updates to improve performance.

4

Strong Consistency

Assumption: MANDATORY for seat commit to prevent oversells.



High-Level Architecture

The system uses a microservices approach, decoupled for independent scaling and maintainability.

CDN

Serves static assets (images, JS)

API Gateway

Single entry point, handles rate limiting

Services

Events (Read), Inventory (Write), Orders

Cache & Queues

Redis for sessions, RabbitMQ for async processing

Core Sequence: Seat Reservation

1

1. Request

User sends POST /reserve with a seat_id

2

2. Lock Attempt

Inventory Service attempts atomic SETNX in Redis with 10-min expiry

3

3. Success

If lock succeeds, service returns 200 OK with reservation_id

4

4. Fail / Conflict

If lock fails (seat taken), service returns 409 Conflict. No oversell occurs

Read Path vs. Write Path

Read Path (Browsing)

Goal: High availability, low latency

Traffic: ~300,000+ RPS

Solution: Served almost entirely by CDN and Redis Read Cache. Database is protected.

Write Path (Buying)

Goal: Strong consistency, high throughput

Traffic: ~10,000+ RPS (spiky)

Solution: Handled by Inventory Service using Redis Atomic Locks and Async Message Queue.

Observability & Resiliency

Logs

Structured JSON logs answer: "What went wrong for user_id 123?"

Metrics

Dashboards (Prometheus) answer: "How many tickets/sec are we selling?"

Traces

End-to-end request tracing answers: "Why is the checkout API slow?"

Circuit Breakers

Stop cascading failures with exponential backoff and aggressive timeouts (~500ms).

Key Trade-Offs & SLOs

99.95%

Availability Target

System uptime guarantee with no single point of failure

<2s

Checkout Latency

p99 latency target for optimal user experience

10K+

Reservations/Sec

Peak throughput capacity during flash sales

Consistency > Performance

Strong consistency (SETNX) guarantees no oversells—correctness over speed.

Stability > UX

Virtual Waiting Room protects system. 2-minute wait beats 503 errors.

Decoupling > Immediacy

Async Queue for checkout gives instant UI response, email follows seconds later.

Thank You...