System Design: Scalable Event Ticketing

A High-Availability, High-Concurrency Solution

The Challenge: Flash Sales

Design a system to sell tickets for popular events with extremely high, spiky traffic, ensuring low latency and preventing oversells.

Part 1: The Requirements

Understanding the stakeholders, goals, and constraints.

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

Functional (What it does)

- Browse and search for events.
- View interactive seat maps.
- Select seats for a 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

- **Constraint:** Must use a 3rd-party payment gateway (e.g., Stripe).
- **Constraint:** System must be designed for malicious bots (requires rate limiting).
- **Assumption: Eventual Consistency** is acceptable for event *listings* and updates.
- **Assumption: Strong Consistency** is MANDATORY for *seat commit* to prevent oversells.

Part 2: Diagrams & Architecture

Visualizing the solution.

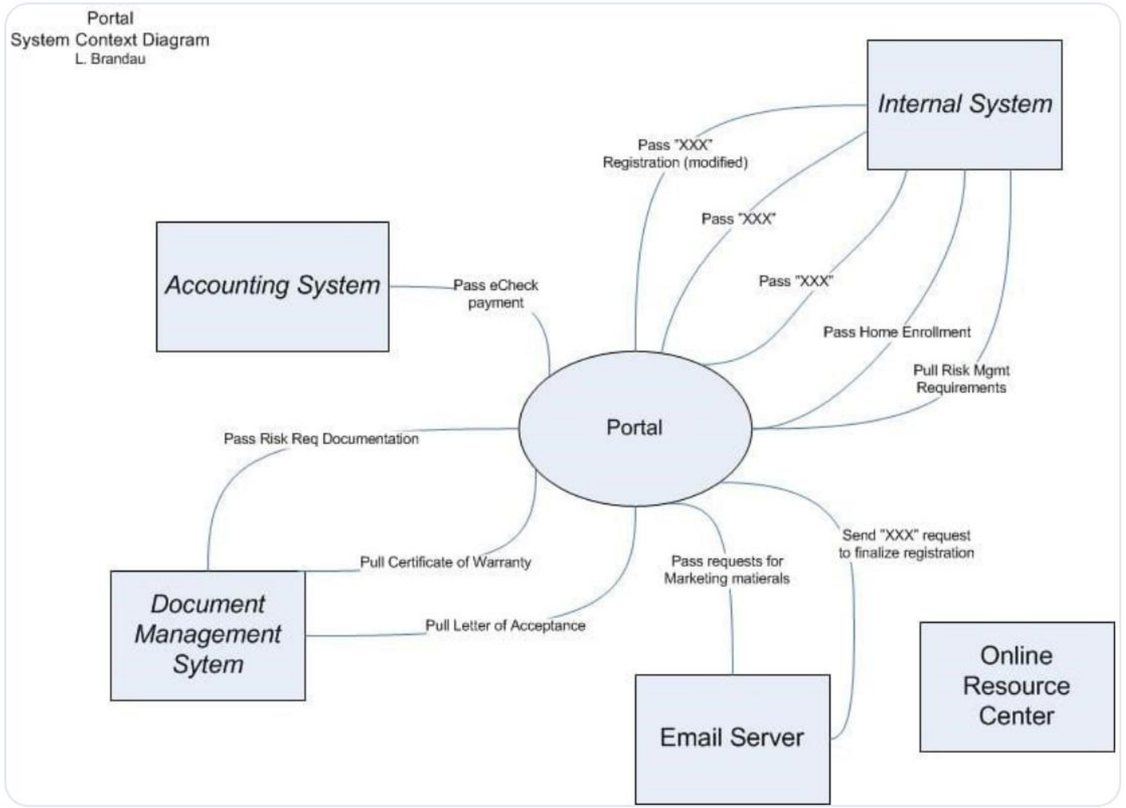
System Context & Use Case

System Context

Defines the system boundary. Shows actors (Buyers, Organizers) interacting with our **Ticketing System**, which in turn connects to external systems like **Payment Gateways** and **Email Services**.

Use Case

Defines the primary actions:
- **Buyers** can: Browse, Reserve, Pay.
- **Organizers** can: Create Event, View Reports.

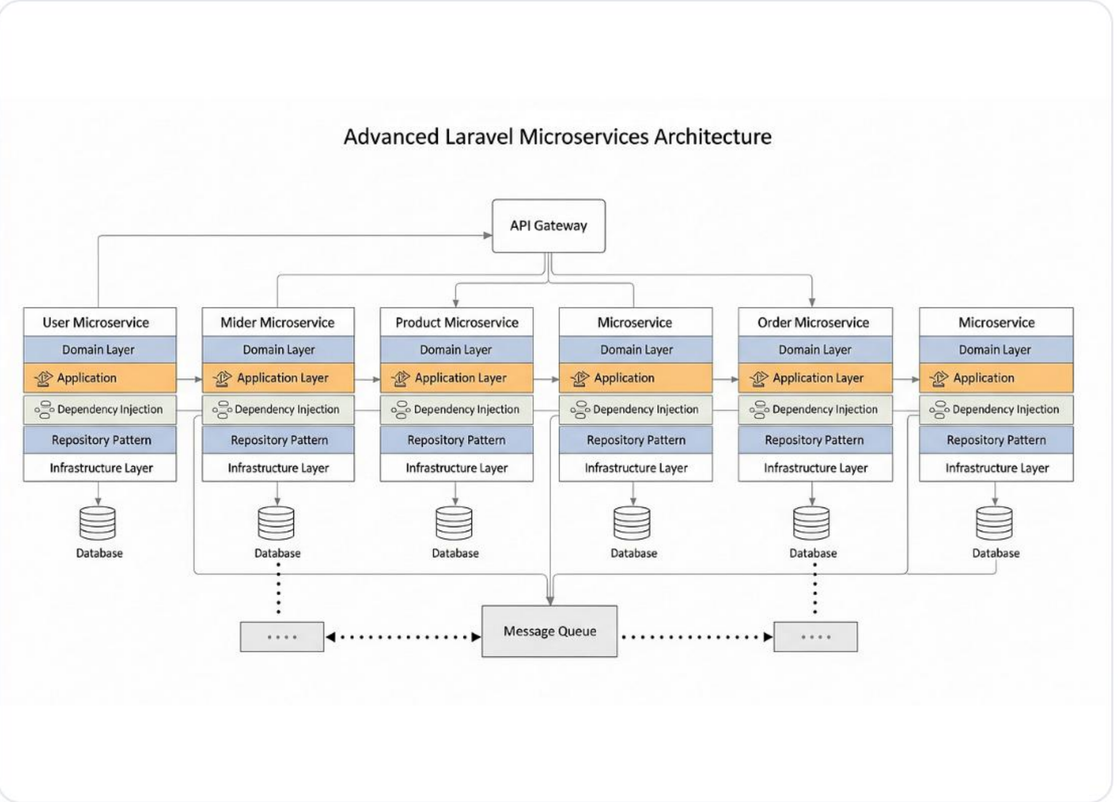


High-Level Architecture

Microservices Approach

The system is 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 (Redis):** For session, read-data, and atomic locks.
- **Queues (RabbitMQ):** Decouples checkout for async processing.
- **Databases (Postgres):** Stores persistent data.



Core Sequence: Seat Reservation

1. Request

User sends `POST /reserve` with a `seat_id`.

2. Lock Attempt

`Inventory Service` attempts an atomic `SETNX` in **Redis** with a 10-min expiry.

3. Success

If lock succeeds, service returns `200 OK` with a `reservation_id`.

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 a **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 an **Async Message Queue**.

Part 3: Engineering Notes

The technical deep dive.

Capacity Sizing & API Contracts

Capacity Estimates

- **Read Path (300k+ rps):** This is the "easy" path, horizontally scalable with CDN/Cache.
- **Write Path (10k+ rps):** This is the bottleneck. System is sized for this, with the Inventory Service & Redis cluster being the key components.

Core API Contracts

- `POST /v1/reserve`
  - **Success:** `200 OK` (Seat locked)
  - **Fail:** `409 Conflict` (Seat taken)
- `POST /v1/checkout`
  - **Success:** `222 Accepted`
  - *Returns 202 to signal asynchronous processing via the message queue.*

Caching Plan & Resiliency

Caching Plan (3 Layers)

- **L1 (CDN):** Static assets (JS, CSS, images).
- **L2 (Read Cache):** "Hot" event data from DB (Redis, 30s TTL).
- **L3 (State Cache):** Seat locks (`event:seat -> user_id`). This is the *source of truth* for locks.

Resiliency & Failure

- **Circuit Breakers:** Stop cascading failures (e.g., if Payment Gateway is down, we stop sending requests).
- **Retries:** Clients use exponential backoff with jitter.
- **Timeouts:** Aggressive internal timeouts (~500ms) to fail fast.

Observability (Logs, Metrics, Traces)

Logs

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

Metrics

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

Traces

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

Part 4: Quality & Trade-Offs

Meeting targets and making hard decisions.

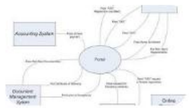
SLOs & Key Trade-Offs

- **SLO Targets:** 99.95% Availability & p99 Checkout Latency < 2 seconds.
- **Trade-Off 1 (Consistency > Performance):** We use **Strong Consistency** (`SETNX`) to *guarantee* no oversells. This correctness is more important than a few milliseconds.
- **Trade-Off 2 (Stability > UX):** A **Virtual Waiting Room** is used to protect the system. A 2-minute wait is a better user experience than a 503 error.
- **Trade-Off 3 (Decoupling > Immediacy):** We use an **Async Queue** for checkout. The user gets an "instant" UI response, even if the confirmation email is delayed by seconds.

Questions?

Thank You

Image Sources



https://www.bridging-the-gap.com/wp-content/uploads/2023/10/System-Context-Diagram-Example.jpg

Source: www.bridging-the-gap.com

https://miro.medium.com/v2/resize:fit:1400/1*z0i1qFCaYAeCiqJoRAdqvQ.png

Source: medium.com