High-Load Ticketing System

Microservices Architecture Documentation

Generated: 2025-11-16 18:43:21

Table of Contents

1. System Overview

The High-Load Ticketing System is a distributed microservices architecture designed to handle high-concurrency ticket booking operations. The system employs a hybrid communication pattern combining synchronous REST API calls for critical operations and asynchronous event-driven architecture using RabbitMQ for eventual consistency updates.

Component	Purpose	Port
API Gateway	Central entry point, routing, authentication	8081
Auth Service	User authentication, JWT tokens, role management	5073
Event Service	Event/session/seat management	8000
Booking Service	Reservation & ticket management	8001
RabbitMQ	Asynchronous message broker	5672/15672
Redis	Distributed locking & caching	6379
PostgreSQL	Primary data storage (multiple instances)	Various

2. Complete User Flow (Green Path)

This section describes a successful end-to-end ticket booking journey from a user's perspective, demonstrating the complete integration of all services.

Step 1: Browse Available Seats

Service: Event Service (Port 8000)

Endpoint: `GET /api/v1/sessions/{session_id}/seat-map`

Description: User views the seat map for a specific event session. The system returns all seats with their status (available, reserved, sold) and pricing information.

Result: Returns seat map with available seats marked as status: "available"

Step 2: Create Reservation

Service: Booking Service (Port 8001)

Endpoint: `POST /api/v1/reservations`

Description: User selects seats and creates a reservation. The system performs multiple backend operations: (1) Validates seat availability via REST call to Event Service, (2) Acquires distributed locks in Redis to prevent double-booking, (3) Fetches real-time pricing from Event Service, (4) Reserves seats via REST API, (5) Creates reservation in database with 10-minute expiry timer.

Result: Returns reservation ID with status "active" and 10-minute countdown timer

Step 3: Verify Seat Reserved

Service: Event Service (Port 8000)

Endpoint: `GET /api/v1/sessions/{session_id}/seat-map`

Description: System or user verifies that selected seats are now marked as "reserved" and temporarily locked.

Result: Seat status changed from "available" to "reserved"

Step 4: Complete Payment & Confirm

Service: Booking Service (Port 8001)

Endpoint: `POST /api/v1/reservations/{id}/confirm`

Description: User completes payment (mock or real gateway). System updates reservation status to "confirmed", generates tickets with QR codes, and publishes a "reservation.confirmed" event to RabbitMQ for asynchronous processing.

Result: Reservation confirmed, tickets generated with QR codes

Step 5: Asynchronous Seat Update

Service: Event Service Consumer (Background)

Endpoint: `RabbitMQ Event Processing`

Description: Event Service consumes the "reservation.confirmed" message from RabbitMQ queue and updates seat status to "sold" in the database. This happens asynchronously (typically < 5 seconds).

Result: Seat status updated to "sold" in Event Service database

Step 6: Retrieve Tickets

Service: Booking Service (Port 8001)

Endpoint: `GET /api/v1/tickets/user/{user_id}`

Description: User retrieves their purchased tickets with embedded QR codes for venue entry.

Result: Returns ticket details with base64-encoded QR code image

3. Service Documentation

3.1. API Gateway

The API Gateway serves as the single entry point to the ticketing system, exposed to the public internet. It implements authentication, authorization, and request routing to downstream microservices.

Primary Responsibilities:

• Request routing to appropriate microservices based on URL patterns (/:service/*proxyPath)

• JWT token validation and authentication middleware

• Role-Based Access Control (RBAC) enforcement

• CORS policy management

• API endpoint configuration management stored in PostgreSQL

Technology Stack:

Technology	Version/Details	Purpose
Language	Go 1.24.4	Core programming language
Framework	Gin Web Framework	HTTP server and routing
Database	PostgreSQL 16	Endpoint configuration storage
Authentication	JWT (golang-jwt/jwt)	Token-based authentication
ORM/Driver	pgx/v5	PostgreSQL driver
Port	8081	HTTP service port

Key Features:

• Proxy pattern: Routes requests to backend services dynamically

• JWT signing key: Shared with Auth Service for token verification

• Swagger/OpenAPI documentation at /swagger/*

• Health check endpoint at /health

• Endpoint management API at /api/v1/endpoint*

3.2. Authentication Service

A dedicated authentication and authorization microservice built with ASP.NET Core. It manages user registration, login, JWT token generation, refresh token rotation, and role-based access control.

Primary Responsibilities:

• User registration and credential management

• JWT access token generation (15-minute expiry)

• Refresh token rotation (30-day expiry)

• Role assignment and permission management

• Structured logging to Elasticsearch for audit trails

Technology Stack:

Technology	Version/Details	Purpose
Language	C# / .NET	Core programming language
Framework	ASP.NET Core 8	Web API framework
Database	PostgreSQL 15	User and role data storage
ORM	Entity Framework Core	Database access and migrations
Logging	Serilog + Elasticsearch	Structured JSON logging
Monitoring	Kibana	Log visualization and analysis
Port	5073 (external)	HTTP service port

API Endpoints:

• POST /auth/register - User registration with email/password

• POST /auth/login - Authentication returning access + refresh tokens

• POST /auth/refresh - Rotate refresh token and issue new access token

• POST /auth/assign-role - Assign role to user (admin operation)

Security Features:

• JWT signing with strong secret key (configurable)

• Issuer and audience validation

• Password hashing (Entity Framework Identity)

• Automatic database migrations on container startup

• Elasticsearch audit logging for compliance

3.3. Event Service

The Event Service manages the core event catalog including events (concerts, sports, movies), sessions (scheduled occurrences), venues, halls, seats, and real-time seat availability. It follows a layered architecture pattern with strict separation of concerns.

Primary Responsibilities:

• CRUD operations for events, locations, halls, and seats

• Session scheduling with time slot validation

• Real-time seat availability tracking

• Internal REST API for Booking Service integration

• RabbitMQ consumer for asynchronous seat status updates

• Multi-language content support (EN, KZ, RU)

Technology Stack:

Technology	Version	Purpose
Language	Python 3.11+	Core programming language
Framework	FastAPI 0.119.0	Async web framework
Database	PostgreSQL 15	Event and session data
ORM	SQLAlchemy 2.0.44 (sync)	Database ORM
Message Queue	RabbitMQ + aio-pika	Event consumer
Cache	Redis 5.0.1	Optional caching
Port	8000	HTTP service port

Architecture Pattern: Layered Architecture

• API Layer (app/api/) - HTTP request/response handling

• Service Layer (app/services/) - Business logic and orchestration

• Repository Layer (app/repositories/) - Database operations

• Models Layer (app/models/) - Domain models and Pydantic schemas

Key API Endpoints:

Public API:

• GET /api/v1/sessions - List all sessions with pagination

• GET /api/v1/sessions/{id} - Get session details

• GET /api/v1/sessions/{id}/seat-map - Get seat availability

Internal API (for Booking Service):

• POST /api/internal/sessions/{id}/seats/check-availability

• POST /api/internal/sessions/{id}/seats/reserve

- POST /api/internal/sessions/{id}/seats/release
- POST /api/internal/sessions/{id}/seats/confirm
- GET /api/internal/sessions/{id}/pricing

3.4. Booking Service

The Booking Service handles the ticket reservation and purchase flow. It implements distributed locking to prevent double-booking, manages 10-minute reservation timeouts, processes payments, generates QR-coded tickets, and publishes events for asynchronous processing.

Primary Responsibilities:

• Seat reservation with distributed Redis locks

• 10-minute reservation timeout with automatic expiry

• Payment processing (mock or real gateway integration)

• QR code generation for tickets

• RabbitMQ event publishing for status updates

• Background worker for expired reservation cleanup

Technology Stack:

Technology	Version	Purpose
Language	Python 3.11+	Core programming language
Framework	FastAPI 0.104.1	Async web framework
Database	PostgreSQL 16 (AsyncPG)	Reservation/ticket storage
ORM	SQLAlchemy 2.0.23 (async)	Async database operations
Cache/Lock	Redis 7	Distributed locking, caching
Message Queue	RabbitMQ + aio-pika	Event publishing
Job Scheduler	APScheduler 3.10.4	Background expiry worker
Port	8001	HTTP service port

Distributed Locking Mechanism:

The service uses Redis SETNX (SET if Not eXists) commands to acquire atomic locks on seats. Lock key pattern: booking:lock:{session_id}:{seat_id}. Locks expire automatically after 10 minutes (600 seconds).

Key API Endpoints:

• POST /api/v1/reservations - Create new reservation

• GET /api/v1/reservations/{id} - Get reservation details

• POST /api/v1/reservations/{id}/confirm - Confirm after payment

• DELETE /api/v1/reservations/{id} - Cancel reservation

• GET /api/v1/tickets/user/{user_id} - Get user's tickets

• GET /api/v1/tickets/{id}/qr - Get QR code image

Background Worker:

A separate worker process runs every 30 seconds to query expired reservations (expiry_time < NOW()) and processes them in batches of 100. It updates status to 'expired', releases Redis locks, and publishes 'reservation.expired' events to RabbitMQ.

4. Service Communication Patterns

The system employs a hybrid communication pattern combining synchronous REST API calls for critical path operations requiring immediate feedback, and asynchronous event-driven messaging for eventual consistency updates.

Pattern	Use Case	Latency	Example
REST (Sync)	Seat validation, pricing, critical operations	< 100ms	Booking → Event Service
RabbitMQ (Async)	Status updates, notifications, analytics	< 5s	Booking → Event Service
JWT Auth	API Gateway authentication	< 10ms	Gateway → All Services
Redis Lock	Distributed seat locking	< 50ms	Booking Service

Detailed Communication Flows:

1. Reservation Creation (Synchronous REST):

a. User → Booking Service: POST /api/v1/reservations

b. Booking → Event Service: POST /api/internal/sessions/{id}/seats/check-availability

c. Booking → Redis: SETNX booking:lock:{session_id}:{seat_id}

d. Booking → Event Service: GET /api/internal/sessions/{id}/pricing

e. Booking → Event Service: POST /api/internal/sessions/{id}/seats/reserve

f. Booking → PostgreSQL: INSERT reservation record

g. Booking → User: Return reservation with 10-min timer

2. Payment Confirmation (Hybrid: REST + RabbitMQ):

a. User → Booking Service: POST /api/v1/reservations/{id}/confirm

b. Booking → PostgreSQL: UPDATE reservation status = 'confirmed'

c. Booking → PostgreSQL: INSERT ticket with QR code

d. Booking → RabbitMQ: PUBLISH 'reservation.confirmed' event

e. Booking → User: Return confirmation (< 300ms)

f. RabbitMQ → Event Service Consumer: DELIVER message

g. Event Service → PostgreSQL: UPDATE seat status = 'sold'

3. Authentication Flow:

a. User → API Gateway: Request with credentials

b. Gateway → Auth Service: Forward authentication request

c. Auth Service → PostgreSQL: Validate credentials

d. Auth Service → User: Return JWT access + refresh tokens

e. User → API Gateway: Subsequent requests with JWT in header

f. Gateway: Validate JWT locally (no Auth Service call needed)

5. Technology Stack Summary

Category	Technologies	Services Using
Languages	Go, C#/.NET, Python	Gateway, Auth, Event, Booking
Web Frameworks	Gin, ASP.NET Core, FastAPI	Gateway, Auth, Event/Booking
Databases	PostgreSQL 15-16	All services
Caching/Locking	Redis 7	Booking, Event (optional)
Message Broker	RabbitMQ 3	Event, Booking
Authentication	JWT (multiple implementations)	Gateway, Auth
Logging	Serilog, Elasticsearch, Kibana	Auth Service
ORM/Drivers	pgx, EF Core, SQLAlchemy	Gateway, Auth, Event, Booking
Containerization	Docker, Docker Compose	All services
API Documentation	Swagger/OpenAPI	All services

6. Infrastructure & Deployment

The system uses Docker containers orchestrated by Docker Compose for local development and testing. All services are connected via a shared Docker network (ticketing_network) enabling service-to-service communication using container names as hostnames.

Docker Network Architecture:

• Network Name: ticketing_network (bridge mode, external: true)

• Allows inter-service communication via container names

• Example: booking-service can reach event-service:8000

• Services expose ports to host for external access

Shared Infrastructure Components:

• RabbitMQ: Shared message broker (container: ticketing-rabbitmq)

• Redis: Shared cache and lock manager (container: ticketing-redis)

• PostgreSQL: Multiple isolated database instances per service

Deployment Scripts:

• start.sh - Initializes network and starts all services

• stop.sh - Graceful shutdown with optional volume cleanup

• test_integration.sh - Automated integration testing

• seed_and_test.sh - Seed test data and run user flow tests

Health Checks:

All services implement health check endpoints and Docker health checks to ensure proper startup sequencing and runtime monitoring. Services depend on database health checks before starting.

Port Mapping Summary:

Service	Internal Port	External Port
API Gateway	8081	8081
Auth Service	8080	5073
Event Service	8000	8000
Booking Service	8001	8001
RabbitMQ AMQP	5672	5672
RabbitMQ Management	15672	15672
Redis	6379	6379

Event DB (PostgreSQL)	5432	5433
Booking DB (PostgreSQL)	5432	5434
Auth DB (PostgreSQL)	5432	5432
Gateway DB (PostgreSQL)	5432	5435

Production Considerations:

• Replace mock payment gateway with real payment processor
• Implement API rate limiting and throttling
• Add distributed tracing (e.g., Jaeger) for request correlation
• Deploy to Kubernetes for horizontal scaling and high availability
• Implement circuit breakers for resilient inter-service communication
• Use managed database services (AWS RDS, Google Cloud SQL)
• Configure SSL/TLS for all external endpoints
• Implement comprehensive monitoring with Prometheus and Grafana
• Set up centralized logging aggregation
• Implement backup and disaster recovery procedures