Reservation System Architecture (Using Express.js)

Status: Proposed
Author: Elias Lezcano
Created: 12/10/2025

1. Executive Summary & Requirements

1.1 Problem Statement and business challenges

Stellar Hotels requires an ergonomic architecture design which also gives it the flexibility to scale the infrastructure easily.

A monolithic architecture can be suitable for a demand of 1,000 bookings per day, but as the main goal is to support a load of +50,000 bookings per day, then we must use a more distributed architecture, leading the development to a **microservices architecture** with a good separation of concerns.

The core technical challenge is balancing the need for strong transactional consistency during the booking process (room hold, payment) with the requirement for massive scalability and 99.9% uptime during high-demand periods. This is further complicated by the need to integrate complex, dynamic pricing rules and external payment gateways.

1.2 Proposed architectural approach

The proposed solution is **Microservices Architecture** leveraging **Domain-Driven Design (DDD)** for defining bounded contexts and **Hexagonal Architecture (Ports and Adapters)** for ensuring a clean, testable separation between core business logic (Domain) and infrastructure concerns (Adapters).

Communication will be handled through the combination of synchronous REST APIs for immediate request/response needs and asynchronous, event-driven model for resilience and decoupling long-running processes.

By applying resilience patterns (Circuit Breakers, Retries) and utilizing the Orchestration-based Saga pattern for distributed transactions, we will achieve one requirement which is reliability.

1.3 Key technical and business requirements

Category	Requirement	Description
Business	**Scalable Booking Volume**	Handle a target of 50,000+ bookings per day without performance degradation.
Business	Dynamic Pricing Engine	Support complex, rule-based pricing that can adapt to seasonality, demand, and promotions.
Business	Extensible Payment Options	Integrate with multiple payment gateways (e.g., Stripe, PayPal) seamlessly.
Technical	**High Availability**	Achieve 99.9% uptime, especially during high-traffic periods.
Technical	Low Latency	Critical API endpoints (e.g., price quoting, booking creation) must respond in under 200ms (p99).
Technical	Fault Tolerance	The system must be resilient to partial failures, especially from external service dependencies.
Technical	Maintainability	The architecture must support parallel development and independent deployment of its components.
Technical	Observability	Full visibility into system

		health through metrics, structured logging, and distributed tracing.

1.4 Success metrics and constraints.

Success metrics:

- API Latency (p99) < 200ms for core booking flow.
- System Uptime ≥ 99.9%.
- Error Rate < 0.1% for all API endpoints.
- Linear scalability of service increasing load.

Constraints:

- The system must be PCI DSS compliant for handling payment information.
- Initial technology stack will be centered around **TypeScript/NodeJS**.

To achieve the 99.9% uptime, the system requires a multi-zone, highly available infrastructure.

2. System Architecture

2.1. Service Architecture (Hexagonal Design)

The system will be decomposed into four core services, each with a clearly defined domain and hexagonal boundary. All services will be implemented using **Express.js**, focusing on separation of concerns through explicit application of the Hexagonal Architecture pattern.

A. Reservation Service

The orchestrator of the booking lifecycle.

- **Core Business Responsibilities:** Manages the state of a booking (e.g., PENDING, CONFIRMED, CANCELLED). Validates booking requests. Coordinates with other services to fulfill a booking request.
- **Primary Ports (Incoming):** REST API Port (Exposes endpoints for creating, retrieving, and canceling reservations). Event Consumer Port (Listens for events like PaymentConfirmed or PaymentFailed).
- **Secondary Ports (Outgoing):** Room Service Port, Pricing Service Port, Payment Service Port, Reservation Repository Port, Event Publisher Port.
- **Adapters: Primary: Express Router** (REST API), RabbitMQ Consumer. **Secondary:** gRPC Client (for Room/Pricing services), PostgreSQL ReservationRepository, RabbitMQ Event Publisher.

B. Room Service (Inventory Management)

The single source of truth for room availability.

- **Core Business Responsibilities:** Manages hotel room inventory. Provides room availability information for given date ranges. Updates inventory based on new reservations or cancellations.
- **Primary Ports (Incoming):** gRPC API Port (Exposes high-performance, internal-facing endpoints for CheckAvailability and UpdateInventory).
- **Secondary Ports (Outgoing):** Room Repository Port, Cache Port.
- **Adapters: Primary:** gRPC Server Controller. **Secondary:** PostgreSQL RoomRepository, Redis CacheRepository.

C. Pricing Service

Handles all dynamic pricing calculations.

- **Core Business Responsibilities:** Calculates the price for a given room type, date range, and occupancy. Applies discounts, promotions, and dynamic rules.
- **Primary Ports (Incoming):** gRPC API Port (Exposes an internal CalculatePrice endpoint).
- **Secondary Ports (Outgoing):** Pricing Rule Repository Port, Market Data Port (Future).
- **Adapters: Primary:** gRPC Server Controller. **Secondary:** PostgreSQL PricingRuleRepository, External API Client.

D. Payment Service

Encapsulates all interactions with payment providers.

- **Core Business Responsibilities:** Processes payments for reservations. Handles refunds and chargebacks. Manages different payment provider integrations.
- **Primary Ports (Incoming):** Event Consumer Port (Listens for BookingCreated events to trigger payment processing).
- **Secondary Ports (Outgoing):** Payment Gateway Port, Payment Repository Port, Event Publisher Port.
- **Adapters: Primary:** RabbitMQ Consumer. **Secondary:** StripeAdapter, PayPalAdapter, PostgreSQL PaymentRepository, RabbitMQ Event Publisher.

2.2. Communication Architecture

Synchronous Communication: Booking Initiation Flow

The initial part of the booking is synchronous to provide immediate feedback to the user.

- **Client API Gateway:** User sends a POST /reservations request with booking details.
- **API Gateway Reservation Service:** The request is forwarded.
- **Reservation Service Room Service (gRPC):** Reservation Service calls CheckAvailability on the Room Service.
- **Reservation Service Pricing Service (gRPC):** Reservation Service calls CalculatePrice on the Pricing Service.

- **Reservation Service Client:** If successful, the Reservation Service creates a PENDING booking, publishes a BookingCreated event, and returns a **202 Accepted** response to the client with the booking ID.
- **Error Handling:** Standard HTTP status codes are used (e.g., 400 Bad Request, 409 Conflict, 503 Service Unavailable).
- **SLAs:** The entire synchronous flow must complete with latency under .

Asynchronous Communication: Payment and Confirmation

Decoupling payment processing ensures the system remains responsive and resilient.

- **Event Bus (RabbitMQ):** The Reservation Service publishes a BookingCreated event.
- **Payment Service Consumer:** The Payment Service consumes the event.
- **Payment Service External Gateway:** It calls the appropriate payment provider (e.g., Stripe) to process the payment.
- **Event Bus (RabbitMQ):** The Payment Service publishes a PaymentConfirmed or PaymentFailed event.
- **Reservation Service Consumer:** The Reservation Service consumes the payment status event and updates the booking to **CONFIRMED** or **CANCELLED**.
- **Notification Service (Future):** A separate Notification service can also listen to these events.

3. Scalability & Reliability Strategy

3.1. Scalability Design

- **Horizontal Scaling:** All services will be containerized using Docker and deployed on a Kubernetes cluster. Horizontal Pod Autoscaler (HPA) will scale the number of service instances based on utilization.
- **Database Scaling:** We will use a managed PostgreSQL service.
- **Read/Write Separation:** Read-heavy services will utilize one or more **read replicas**.
- **Connection Pooling:** PgBouncer will be used to manage database connections efficiently.
- **Caching Strategy:** A distributed Redis cluster will be used (e.g., CACHE-ASIDE pattern in Room Service, memoization in Pricing Service).
- **Load Balancing:** An API Gateway (e.g., Nginx Ingress Controller) will provide L7 load balancing.

3.2. Reliability Patterns

- **Retry Policies:** For transient network errors during inter-service gRPC calls, a retry policy with exponential backoff and jitter will be implemented.
- **Circuit Breaker Pattern:** Critical for the Payment Service. External payment gateway adapters will be wrapped in a circuit breaker (e.g., using the opossum library) to prevent cascading failures.
- **Health Monitoring:** Each service will expose a /health endpoint for Kubernetes liveness and readiness probes. Prometheus will scrape metrics for monitoring and

alerting.

- **Timeouts:** Aggressive but realistic timeouts will be configured for all network requests.

4. Data Architecture

- **Database per Service:** Each microservice will own and manage its own private database, ensuring loose coupling and independent evolution of data schemas.
- **Data Consistency:**
  - **Strong Consistency:** Guaranteed within the boundary of a single service via standard ACID database transactions.
  - **Eventual Consistency:** Used across services. The **Saga pattern** will manage the distributed transaction of a booking (Reservation Payment Confirmation), with compensating events for failures.
- **Performance Optimizations:** Judicious use of indexing, materialized views for read-heavy queries, and offloading analytical queries to read replicas.

5. Technology Stack Justification

Component	Technology	Justification
Language/Runtime	TypeScript / Node.js	Provides strong typing to reduce runtime errors. The vast NPM ecosystem and non-blocking I/O model make it excellent for building high-throughput, I/O-bound microservices.
Framework	**Express.js (with TypeScript)**	**Decision Rationale:** Express.js is chosen for its **minimalist, unopinionated core**, which grants the team **maximum flexibility and autonomy**. Unlike opinionated frameworks like NestJS, Express.js does **not force a specific architectural pattern**, allowing us to be **decoupled** from

		framework decisions and implement the Hexagonal Architecture and DDD boundaries with precision. This approach avoids unnecessary framework abstractions, minimizes boilerplate, and ensures the core business logic remains independent of the delivery mechanism, supporting better **long-term maintainability** and **performance tuning** in a distributed microservices environment.
Database	PostgreSQL	A mature, reliable, and feature-rich open-source relational database. Its support for ACID transactions and scalability with read replicas makes it a strong choice.
Caching	Redis	A high-performance in-memory data store, ideal for distributed caching and as a simple pub/sub message broker.
Messaging/Events	RabbitMQ	Provides reliable delivery, flexible routing, and strong message guarantees, ideal for transactional workflows and ensuring consistency and fault tolerance.
Infrastructure	Docker & Kubernetes	The industry standard for containerization and

		orchestration. Provides automated deployment, scaling, and operational management.
API Gateway	Nginx Ingress	A powerful, high-performance solution for managing external access to services in a Kubernetes cluster, handling routing, SSL termination, and load balancing.