# **Architecture Design Document: Global Event Booking Platform**

## I. Background

The platform is designed to support users in discovering global events, managing seat bookings, and processing secure payments. Key requirements include horizontal scalability to handle peak loads near event dates, ensuring minimal latency for users, data consistency, secure payment processing, and timely notifications.

## II. Architecture Overview

### System Architecture Diagram

### D:\Quyen\board-absurd-prankster (4).jpgSystem Components

**API Gateway**: The API Gateway serves as the single entry point for all client requests, forwarding them to the appropriate services (e.g., Discovery Service, Booking Service) based on the API route.

**Discovery Service**: Handles full-text search on event data using Elasticsearch, a high-performance search engine supporting advanced features like fuzzy search, proximity search, and ranking. New or updated event data in SQL Server is indexed in Elasticsearch for optimal searchability.

**Elasticsearch (Event Indexes)**: Elasticsearch maintains indexes of event data, enabling fast, complex full-text search operations. Event data in SQL Server is periodically indexed in Elasticsearch via scheduled or real-time syncs to keep search results updated.

**Booking Service**: Manages event bookings, including seat reservations and payments. Transactional data (events, seats, bookings) is stored in SQL Server. The Booking Service integrates with the Payment Gateway (Stripe) for secure payment processing and sends booking confirmations through the Notification Service.

**Event Management Service**: Responsible for creating and updating events, including updating SQL Server records and sending event updates to **Discovery Service** for Elasticsearch indexing.

**SQL Server**: Stores event data, seat reservations, and bookings in a structured, relational format. SQL Server manages operations like seat availability checks and booking records. Event data is indexed in Elasticsearch for search optimization.

**Payment Gateway**: Processes payments when users book events, ensuring secure payment handling via third-party providers.

**Notification Service**: After booking confirmation, this service (e.g., Amazon SES for email, Twilio for SMS) sends notifications to users, confirming their bookings.

**User Service**: Manages user data, including profiles and booking history, stored securely in SQL Server. This service handles user-related operations like authentication and profile updates.

**Finance Service**: Integrated with the Booking Service and Payment Gateway to track payments, manage accounting, process refunds, and generate financial reports.

**Reporting/Analytics Service**: Aggregates data from SQL Server and Elasticsearch to produce insights, such as sales trends and customer analytics, for business intelligence.

**SEO Service**: Works alongside the Search/Discovery Service to optimize event pages for search engines, enhancing organic traffic to the platform.

**Marketing Service**: Executes campaigns (email, social media, retargeting) and manages promotions, providing personalized event recommendations and promotion management.

## III. Database Diagram

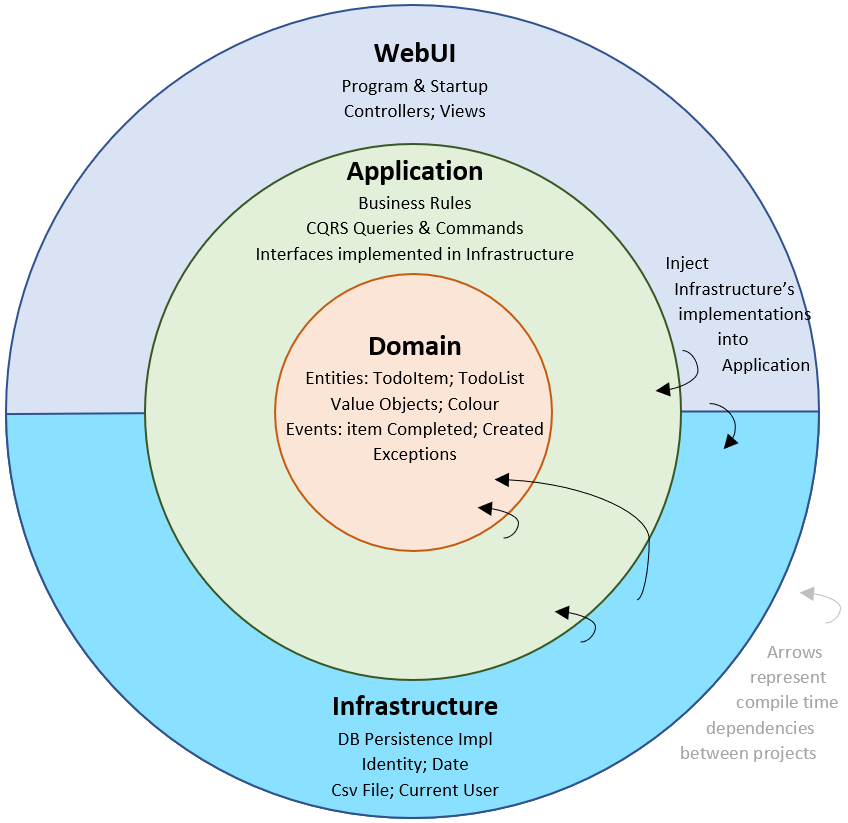
### Database Diagram

### D:\Quyen\database-diagram.png

### Service Layer and Data Model

**Service Layer**: These services will interact with each other via RESTful APIs/async communication(use message broker).

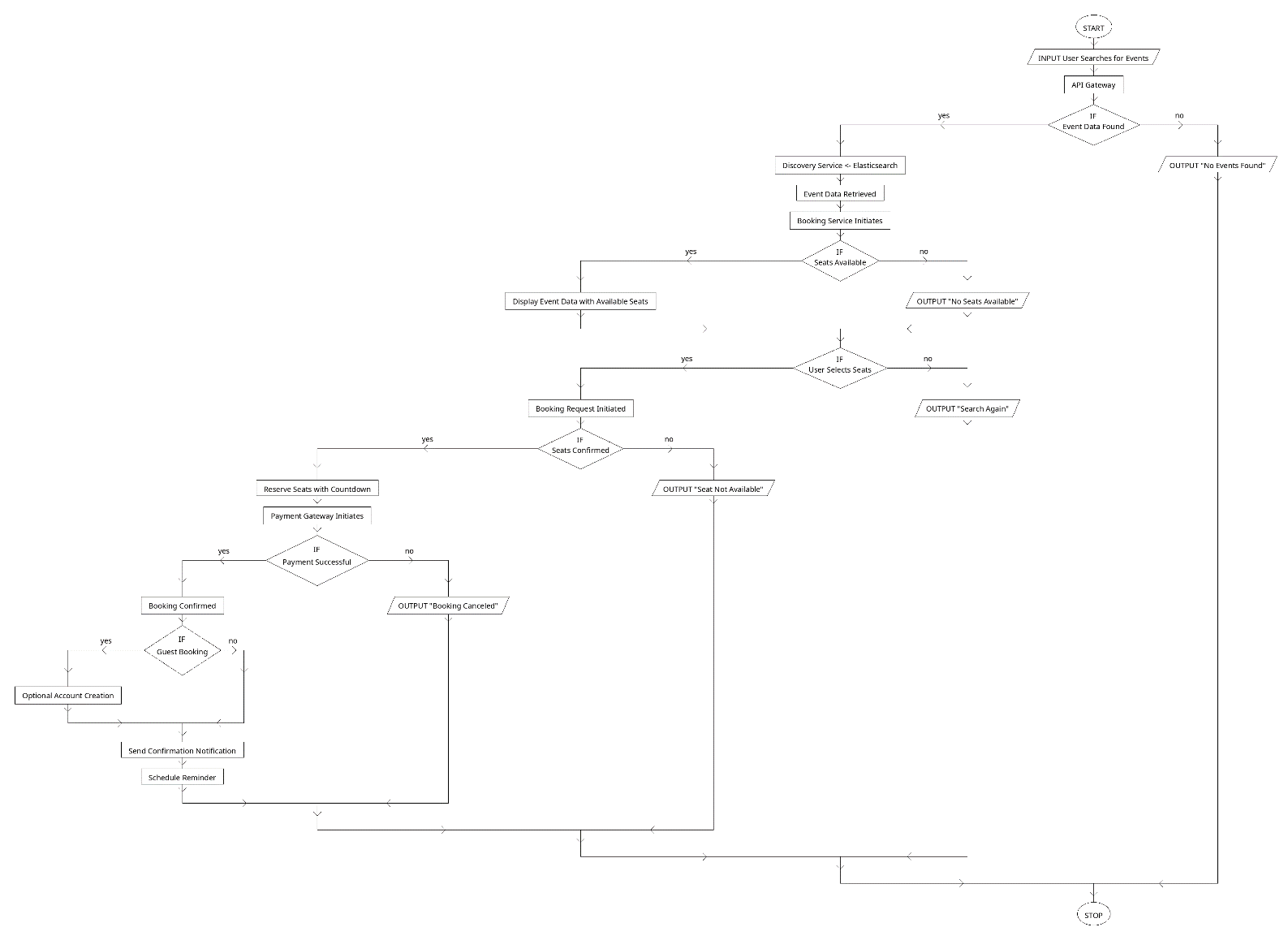
Each services use Clean Architecture itself.



**Data Model**:

1. **Users**: Stores registered users with fields like UserID, Name, Email, PasswordHash, and DateCreated.
2. **GuestBookings**: Tracks bookings by guest users, including GuestBookingID, Name, Email, EventID, SeatID, DateBooked, and Status.
3. **Events**: Holds event details with fields like EventID, Name, Location, Description, Date, TotalTickets, and AvailableTickets.
4. **Seats**: Manages individual seats within an event, with fields like SeatID, EventID, UserID, GuestBookingID, Price, and Status.
5. **Bookings**: Links users or guests to reserved seats with fields like BookingID, UserID, GuestBookingID, PaymentStatus, and BookingDate.
6. **BookingSeats**: Join table linking Bookings to Seats, supporting multiple seats per booking.
7. **Payments**: Tracks payments for bookings, including fields like PaymentID, BookingID, Amount, Status, and Timestamp.
8. **Refunds**: Manages payment refunds with fields like RefundID, PaymentID, Amount, Reason, and Status.
9. **Analytics, Marketing, SEO, and Reports**:
   * **Analytics**: Aggregates booking and event data for reporting.
   * **Marketing**: Stores campaign data.
   * **SEO**: Optimizes event listings with keywords and metadata.
   * **Reports**: Aggregates data for business intelligence.

## IV. User Flow Chart



## V. Technology Stack

**Backend**: .NET Core (with support for other languages and frameworks for microservices)

**Database**:

* SQL Server: Manages transactional data (e.g., bookings, payments) and user data.
* Elasticsearch: Provides a search engine for event data.
* Redis: Used for caching frequently accessed data.

**Message Broker**: RabbitMQ or Apache Kafka for event-driven processing.

**Monitoring and Metrics**: Prometheus and Grafana.

**Logging**: ELK Stack (Elasticsearch, Logstash, Kibana).

**API Gateway**: NGINX/Kong.

**Containerization**:

* Docker: For containerizing services.
* Kubernetes: For orchestration.

**Storage**: AWS S3.

**Security**: IdentityServer4 with OAuth2 for secure authentication and authorization.

**3rd Party Payment Gateway**: Stripe.

**Deployment**: AWS.

## VI. System Pillars

### Scalability

* **Horizontal Scaling**: Stateless services (e.g., Booking, Search, User) can scale horizontally by adding more instances.
* **Auto-scaling**: Uses Docker Swarm, Kubernetes, or cloud platforms (AWS, Azure) for dynamic scaling based on real-time load.
* **Database Sharding**: Enhances performance for high-volume data like event listings.

### Robustness

* **Redundancy**: Ensures uptime through multiple servers in different regions.
* **Auto-scaling**: Uses AWS Auto Scaling or Kubernetes to handle load.
* **Error Handling and Retries**: Implements retry mechanisms, particularly for payment processing.
* **Circuit Breakers**: Prevents overload when dependent services fail.
* **Monitoring and Alerting**: Tools like Prometheus, Grafana, and AWS CloudWatch monitor system health and resolve issues proactively.

### Security

* **Payment Gateway**: Integrates with Stripe, following HTTPS and tokenization standards.
* **Authentication and Authorization**: IdentityServer4 manages secure authentication using OAuth2 and OpenID Connect. JWT tokens ensure tamper-proof authorization.
* **Data Protection**: Encrypts sensitive data in transit and at rest.

### Maintainability

* **Microservices Architecture**: Supports independent updates for each service (e.g., Booking, User, Event Search, Notification).
* **CI/CD**: Pipelines for automated building, testing, and deployment.
* **Logging and Monitoring**: Real-time monitoring with tools like Prometheus and Grafana.

### Handling Peak Hours

* **Auto-scaling**: Services like Booking and Event Search scale up during peak times.
* **Load Balancing**: Distributes traffic evenly across instances.
* **Database Performance**: Uses Elasticsearch for high-throughput searches and SQL Server for transactional data.
* **Caching**: Redis caching of frequently accessed data (e.g., event listings) reduces backend load.
* **Optimistic Locking**: Ensures concurrent seat reservations without conflicts.
* **Retries and Exception Handling**: Limited retries with exception handling for concurrency conflicts.
* **Fraud Prevention**: Rate limiting, payment validation, real-time monitoring, and CAPTCHA.

## VII. Testing and Quality Assurance

* **Unit Testing**: All services will have unit tests.
* **Integration Testing**: End-to-end workflows will be validated with integration tests.
* **Load Testing**: Performed using tools like JMeter to ensure the system can handle peak traffic.

## VIII. Considerations

* **Alternative Full-Text Search**: If advanced full-text search is not required, consider using MongoDB instead of Elasticsearch.
* **Dynamic Schema**: If event data frequently changes, MongoDB can store dynamic data while SQL Server handles Seats and Bookings.
* **Pessimistic Locking Option**: For strict seat booking requirements, Pessimistic Locking can be considered but may limit concurrency.
* **Combined Booking and Payment Flow**: Simplifying the user flow may make booking and payment more atomic.
* **Limit Guest Users**: Reducing guest users simplifies data management but may affect booking volume.

## IX. Conclusion

The architecture of the Global Event Booking Platform is designed to be scalable, secure, and high-performing. By leveraging a microservices architecture and global deployment, the platform can serve users worldwide, providing a seamless booking experience.