Event Ticket Platform Backend

Architecture Overview

The Event Ticket Platform backend is built on a modern, modular architecture using NestJS, a progressive Node.js framework designed for building efficient, reliable, and scalable server-side applications. This document provides a detailed overview of the system architecture, design principles, and module structure.

Table of Contents

Core Technologies

The backend is built using the following core technologies:

- **NestJS**: A TypeScript-based progressive Node.js framework for building enterprise-grade applications

- **Prisma ORM**: Modern database toolkit for type-safe database access
- **MongoDB**: NoSQL database for flexible and scalable data storage
- **Redis**: In-memory data structure store used for caching and session management
- **JWT**: JSON Web Tokens for stateless authentication
- **PayOS**: Integration with payment gateway for processing transactions
- **AWS SDK**: For cloud storage and services integration
- **Jest**: For comprehensive testing

Architectural Principles

The backend architecture follows these key principles:

1. Modularity and Separation of Concerns

Each feature is encapsulated in its own module with clear boundaries and responsibilities. This enables:

- Independent development and testing
- Easier maintenance and updates
- Better code organization and reusability

2. Dependency Injection

NestJS's dependency injection system promotes:

- Loose coupling between components
- Easier unit testing through component mocking
- Simplified service composition and configuration

3. Domain-Driven Design

The application is structured around business domains rather than technical concerns:

- Business logic is isolated in service classes
- Entity models represent the core domain objects
- Repository pattern abstracts data access details

4. RESTful API Design

The API follows REST principles:

- Resource-oriented endpoints
- HTTP methods are used appropriately (GET, POST, PATCH, DELETE)
- Consistent response structures
- Proper status codes

5. Security-First Approach

Security is a primary concern at all levels:

- Authentication using JWT with refresh token rotation
- Role-based access control

- Input validation and sanitization
- Data encryption for sensitive information
- Rate limiting and security headers

System Architecture

The system follows a layered architecture:

1. **Controller Layer**: Handles HTTP requests, validates input, and delegates to services
2. **Service Layer**: Contains business logic and coordinates domain operations
3. **Repository Layer**: Abstracts data access through Prisma ORM
4. **Entity Layer**: Defines the data structures and relationships

High-Level Component Diagram

```
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│   API Layer     │────▶│  Service Layer  │────▶│   Data Layer    │
│  Controllers    │◀────│    Services     │◀────│ Prisma Clients  │
└─────────────────┘     └─────────────────┘     └─────────────────┘
         │                       │                       │
         │                       │                       │
         ▼                       ▼                       ▼
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│     Guards      │     │     DTOs &      │     │    External     │
│ & Interceptors  │     │   Validators    │     │    Services     │
└─────────────────┘     └─────────────────┘     └─────────────────┘
                                                        │
                                                        │
                                                        ▼
                                                ┌─────────────────┐
                                                │   MongoDB       │
                                                │   Redis         │
                                                │   AWS S3        │
                                                │   PayOS         │
                                                └─────────────────┘
```

Database Design

The system uses MongoDB (via Prisma ORM) as its primary database, with the following key entities:

Core Entities

1. **User**: Manages user accounts with role-based permissions (Attendee, Organizer, Admin)
2. **Event**: Stores event information including location, dates, and status
3. **IssuedTicket**: Represents tickets created for events with inventory, class, pricing, and availability status
4. **ClaimedTicket**: Tracks tickets claimed by attendees, with validation details and usage history
5. **Order**: Records purchase transactions for tickets with status tracking and payment links
6. **Payment**: Stores payment information, transaction details, and status updates
7. **Tag**: Categorizes events for filtering and discovery

8. **Review**: Stores user reviews for events and ratings
9. **Venue**: Contains venue information, seating layout, and capacity details
10. **TicketClass**: Defines different ticket tiers, pricing, and benefits for events

Database Schema Highlights

- MongoDB collections with relations managed via Prisma
- Enums for status fields (EventStatus, TicketStatus, OrderStatus)
- Comprehensive indexing strategy for query performance
- Soft deletion pattern for maintaining historical data

Module Structure

The backend is organized into feature modules:

Core Modules

- **AppModule**: Root module that configures application-wide settings
- **PrismaModule**: Provides database connection and repository services
- **ConfigModule**: Manages environment-specific configuration
- **AuthModule**: Handles authentication, authorization, and user management
- **SharedModule**: Contains utilities and services used across multiple modules

Feature Modules

- **EventModule**: Manages events creation, updates, and queries
- **IssuedTicketModule**: Handles ticket creation, inventory, and availability management
- **ClaimedTicketModule**: Manages ticket claiming, validation, and transfer between users
- **OrderModule**: Processes ticket purchase orders and manages order lifecycle
- **PaymentModule**: Integrates with payment gateway for processing transactions
- **TagModule**: Manages event categorization and filtering
- **ReviewModule**: Handles user reviews and ratings for events
- **ImageModule**: Manages image uploads, processing, and storage
- **NotificationModule**: Handles system notifications, emails, and SMS alerts
- **UserModule**: Manages user profiles, preferences, and account settings

Each module follows a consistent structure:

- Controller: Handles HTTP requests
- Service: Contains business logic
- DTO: Defines data transfer objects for validation
- Entities: Defines domain models
- Tests: Contains unit and integration tests

Authentication & Authorization

Authentication Flow

1. **Registration**: Users register with email, username, and password

  - Passwords are hashed using bcrypt
  - Email verification tokens are generated
  - Confirmation emails are sent via event emitters

2. **Login**: Users authenticate with username/password

   - JWT access tokens are issued with short expiry
   - Refresh tokens are issued for token renewal
   - User roles and permissions are encoded in tokens

3. **Token Refresh**: Secure mechanism for extending sessions

   - Rotation-based refresh token strategy
   - Single-use refresh tokens for enhanced security

Authorization Strategy

Role-based access control is implemented with three primary roles:

- **Attendee**: Regular users who can browse events and purchase tickets
- **Organizer**: Can create and manage events and tickets
- **Admin**: Has full system access for management and oversight

Guards enforce authorization:

- **JwtAuthGuard**: Validates JWT tokens
- **RolesGuard**: Enforces role-based access control
- **Custom guards**: For specific business rules

API Endpoints

The API is organized around REST principles with consistent patterns:

Authentication Endpoints

- POST `/auth/register`: Register new user
- POST `/auth/login`: Authenticate user
- POST `/auth/refresh`: Refresh access token
- POST `/auth/logout`: Invalidate tokens

Event Management

- GET/POST/PATCH/DELETE `/events`: Event CRUD operations
- GET `/events/tag/:tagId`: Filter events by tag
- GET `/events/city/:cityId`: Filter events by location

Ticket Management

- GET `/issued-tickets`: Get available tickets for an event

- GET `/issued-tickets/:id`: Get specific ticket details

- POST `/issued-tickets/batch`: Create multiple tickets for an event
- PATCH `/issued-tickets/:id/status`: Update ticket status
- GET `/issued-tickets/stats/:eventId`: Get ticket availability statistics
- GET `/claimed-tickets`: Get user's claimed tickets
- POST `/claimed-tickets/claim`: Claim a purchased ticket
- GET `/claimed-tickets/:id/validate`: Validate ticket for entry
- POST `/claimed-tickets/transfer`: Transfer ticket to another user
- GET `/claimed-tickets/qr/:id`: Generate QR code for ticket

Order Processing

- POST `/orders`: Create new order
- GET `/orders/:id`: Get order details
- PATCH `/orders/:id/status`: Update order status

Payment Integration

- POST `/payments`: Create payment
- GET `/payments/:id`: Get payment status
- POST `/payments/webhook`: Process payment notifications

Detailed API documentation is available in the API Routes Guide.

Payment Processing

The system integrates with PayOS payment gateway:

Payment Flow

1. **Order Creation**: User selects tickets, creating an order with PENDING status
2. **Payment Initiation**: Payment link is generated via PayOS
3. **Payment Processing**: User completes payment on PayOS platform
4. **Webhook Notification**: PayOS notifies the system of payment status
5. **Order Fulfillment**: System updates order status and issues tickets

Key Components

- **PaymentService**: Handles payment gateway integration
- **Webhook Handler**: Processes asynchronous payment notifications
- **Transaction Management**: Ensures data consistency across payment-related operations

Ticket Management

The ticket system represents a core domain in the platform, implemented with a sophisticated architecture that handles the complete lifecycle of tickets from creation to validation.

Ticket Architecture

The ticket management system is split into two primary modules that separate concerns:

1. **IssuedTicketModule**: Manages the supply-side of tickets (creation, inventory, availability)
2. **ClaimedTicketModule**: Manages the demand-side of tickets (purchase, claiming, validation)

Module Relationships

```
┌─────────────────┐      ┌─────────────────┐      ┌─────────────────┐
│   EventModule   │◄─────┤IssuedTicketModule├─────►│   OrderModule   │
└─────────────────┘      └─────────────────┘      └─────────────────┘
         │                        │                        │
         │                        ▼                        │
         │               ┌─────────────────┐              │
         └───────────────┤ClaimedTicketModule │◄────────────┘
                         └─────────────────┘
                                  │
                                  ▼
                         ┌─────────────────┐
                         │ ValidationModule │
                         └─────────────────┘
```

IssuedTicket Module

Core Components

- **IssuedTicketController**: REST API endpoints for ticket management
- **IssuedTicketService**: Business logic for ticket operations
- **IssuedTicketRepository**: Data access layer via Prisma
- **TicketInventoryService**: Manages ticket availability and holds

Design Patterns

- **Repository Pattern**: Abstracts database operations
- **Factory Pattern**: For creating different ticket types
- **Observer Pattern**: For ticket status change notifications
- **Strategy Pattern**: For flexible pricing strategies

Key Features

- **Batch Creation**: Efficient creation of multiple tickets
- **Inventory Management**: Real-time tracking of available tickets
- **Reserved Seating**: Assigned seat mapping with venue layouts
- **Dynamic Pricing**: Support for variable pricing tiers
- **Time-based Availability**: Schedule-based ticket release
- **Hold Management**: Temporary reservation system with timeout

ClaimedTicket Module

Core Components

- **ClaimedTicketController**: REST API for claiming and validation
- **ClaimedTicketService**: Business logic for claiming tickets
- **TicketTransferService**: Handles ticket transfers between users
- **ValidationService**: Verifies ticket authenticity and status

Security Features

- **Cryptographic Signatures**: Prevents ticket forgery
- **One-time Use Codes**: Prevents duplicate usage
- **QR Code Encryption**: Secure ticket representation
- **Time-based Validation**: Tickets only valid during event time
- **Revocation Capability**: Ability to invalidate tickets if needed

Ticket States

1. **AVAILABLE**: Ticket is available for purchase
2. **HELD**: Temporarily reserved during checkout (with timeout mechanism)
3. **PAID**: Purchased but not yet claimed by attendee
4. **CLAIMED**: Associated with a specific attendee (ready for use)
5. **VALIDATED**: Ticket has been used for entry
6. **CANCELLED**: No longer valid (refunded or revoked)
7. **EXPIRED**: Past event date, no longer usable

Ticket Domain Model

```
// Key attributes of the IssuedTicket entity
interface IssuedTicket {
  id: string;
  eventId: string;
  class: string;          // e.g., "VIP", "Standard"
  price: number;
  status: TicketStatus;
  seat?: string;          // Optional for assigned seating
  section?: string;       // Venue section
  row?: string;           // Venue row
  metadata: object;       // Flexible additional data
  createdAt: Date;
  updatedAt: Date;
}

// Key attributes of the ClaimedTicket entity
interface ClaimedTicket {
  id: string;
  issuedTicketId: string;
  userId: string;
  orderId: string;
  validationCode: string; // For entry verification
  validationStatus: ValidationStatus;
  claimedAt: Date;
  validatedAt?: Date;      // When the ticket was used
}
```

Integration Points

- **Event Module**: Tickets are created for specific events
- **Order Module**: Purchases create orders containing multiple tickets
- **Payment Module**: Successful payments trigger ticket status changes
- **User Module**: Associates claimed tickets with specific users
- **Notification Module**: Alerts users about ticket status changes

Advanced Features

- **Seat Selection**: Interactive venue map with real-time availability
- **Ticket Classes**: Multiple tiers with varying prices and benefits
- **Digital Tickets**: QR code generation with cryptographic security
- **Validation System**: Mobile app and kiosk validation at entry points
- **Transfer System**: Secure ticket transfer between users
- **Waitlist Management**: Queue system for sold-out events
- **Dynamic Pricing**: Time-based and demand-based pricing adjustments

Ticket Validation System

The ticket validation system ensures ticket authenticity, prevents fraud, and provides a seamless entry experience at events.

Validation Architecture

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│  Mobile App     │    │ Validation API  │    │ Kiosk Scanner   │
│  Scanner        │───▶│ (NestJS)        │◀───│                 │
└─────────────────┘    └─────────────────┘    └─────────────────┘
                                │
                                ▼
                       ┌─────────────────┐
                       │ Validation      │
                       │ Service         │
                       └─────────────────┘
                                │
                    ┌───────────┴───────────┐
                    │                       │
                    ▼                       ▼
           ┌─────────────────┐    ┌─────────────────┐
           │ ClaimedTicket   │    │ Event           │
           │ Repository      │    │ Repository      │
           └─────────────────┘    └─────────────────┘
                    │                       │
                    └───────────┬───────────┘
```

```
                         │
                ┌────────▼────────┐
                │ Notification    │
                │ Service         │
                └─────────────────┘
```

Validation Methods

1. **QR Code Scanning**: Primary method using mobile app or kiosk scanners
2. **Numeric Code Entry**: Fallback for technology issues
3. **NFC/RFID**: Support for physical wristbands or cards at premium events
4. **Biometric Verification**: Optional enhanced security for high-value events

Security Measures

- **One-time Use Validation**: Prevents duplicate entry with single-use codes
- **Time-window Validation**: Tickets only valid during specific entry periods
- **Cryptographic Signatures**: Digital signatures to prevent forgery
- **Offline Validation Capability**: Validation can work without internet connection
- **Real-time Sync**: Multi-entrance synchronization to prevent entry at different gates

Validation Process Flow

1. **Code Generation**: Secure validation code created upon ticket claim
2. **Pre-Validation**: Optional check-in process before reaching the venue
3. **Entry Validation**: Scanner reads QR code or other validation credential
4. **Verification**: System checks ticket validity, event match, and usage status
5. **Status Update**: Ticket marked as validated in the system
6. **Entry Granted**: Visual and/or audio confirmation of successful validation
7. **Analytics Capture**: Entry data recorded for venue analytics and reporting

Validation Integration Systems

- **Event Management**: Links validation to specific event timing and rules
- **Notification System**: Alerts organizers of validation issues or high traffic
- **Analytics Platform**: Provides real-time entry statistics and flow metrics
- **Security Services**: Identifies suspicious validation patterns

High-Performance Ticket System

The ticketing system is architected to handle high-volume sales scenarios like major concert releases or festival launches, with specific optimizations for performance and reliability.

Performance Optimizations

1. **Database Indexing Strategy**
    - Optimized indexes on ticket status, event ID, and user ID fields
    - Compound indexes for common query patterns

- Sparse indexing for optional ticket attributes

2. **Caching Architecture**

   - Redis cache for ticket availability counts
   - Distributed cache for seat maps
   - Local memory caching for validation codes
   - Cache invalidation patterns for ticket status changes

3. **Queue-Based Processing**

   - Asynchronous ticket creation for large batches
   - Queue-based order processing to handle traffic spikes
   - Rate-limited API endpoints to prevent abuse
   - Priority queues for different ticket operations

Scaling Considerations

1. **Horizontal Scaling**

   - Stateless API design allows for easy node scaling
   - Database read replicas for high-query traffic
   - Sharding strategy for multi-million ticket events

2. **High-Volume Sales**

   - Virtual waiting room implementation
   - Controlled ticket release batches
   - Graceful degradation during peak loads
   - Circuit breakers for dependent services

3. **Monitoring & Recovery**

   - Real-time ticket operation metrics
   - Automated recovery procedures
   - Transaction compensation patterns
   - Self-healing capabilities for failed operations

On-Sale Strategy

For high-demand events, the system employs a sophisticated on-sale strategy:

1. **Pre-Sale Queue**: Users enter a virtual waiting room before tickets go on sale
2. **Controlled Entry**: Gradual admission to the purchase flow to prevent system overload
3. **Hold Management**: Temporary reservation with countdown timer
4. **Completion Window**: Limited time to complete purchase before tickets return to inventory
5. **Auto-Scaling Triggers**: Infrastructure scaling based on queue depth and traffic patterns

Event Management

The Event Module serves as the central component of the platform, representing the core business entity around which all other functionality revolves. It implements a sophisticated architecture for creating, managing, and discovering events.

Event Architecture

```
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│   EventModule   │◄────┤   UserModule    ├────►│    TagModule    │
└────────┬────────┘     └─────────────────┘     └─────────────────┘
         │                                               ▲
         │                                               │
         ▼                                               │
┌─────────────────┐     ┌─────────────────┐     ┌────────┴────────┐
│IssuedTicketModule│◄───┤   ImageModule   ├────►│   SearchModule  │
└─────────────────┘     └────────┬────────┘     └─────────────────┘
                                 │
                                 ▼
                        ┌─────────────────┐
                        │  AWS S3 Service │
                        └─────────────────┘
```

Core Components

Controllers

- **EventController**: Handles HTTP requests for event CRUD operations
- **EventAdminController**: Admin-specific operations for event management
- **EventDiscoveryController**: Specialized endpoints for event discovery and search

Services

- **EventService**: Core business logic for event operations
- **EventValidationService**: Validates event data and business rules
- **EventPublishingService**: Manages the event approval and publishing workflow
- **EventStatsService**: Collects and processes event analytics data

Repositories

- **EventRepository**: Prisma-based data access for event entities
- **EventCacheRepository**: Redis-based caching for high-traffic event data
- **EventSearchRepository**: Elasticsearch integration for advanced search capabilities

Event Domain Model

```
interface Event {
  id: string;
  title: string;
  description: string;
  organizerId: string;
  venueId: string;
  startDate: Date;
  endDate: Date;
  timezone: string;
  status: EventStatus; // DRAFT, PENDING_APPROVAL, PUBLISHED, CANCELLED
  capacity: number;
  featuredImageId: string;
  bannerImageId: string;
  ticketingEnabled: boolean;
  publicUrl: string;
  seoMetadata: {
    title: string;
    description: string;
    keywords: string[];
  };
  settings: {
    showRemainingTickets: boolean;
    enableWaitlist: boolean;
    requireApproval: boolean;
    allowSharing: boolean;
  };
  createdAt: Date;
  updatedAt: Date;
  publishedAt?: Date;
}
```

Design Patterns

- **Builder Pattern**: Implements a fluent API for event creation
- **State Machine**: Manages event lifecycle states and transitions
- **Publisher/Subscriber**: For event-driven notifications of status changes
- **Command Pattern**: For executing and tracking administrative actions
- **Specification Pattern**: For complex query and filtering requirements

Event Lifecycle

1. **Creation**: Organizer creates event draft with basic information
2. **Enrichment**: Additional details, images, and settings are added
3. **Configuration**: Ticket types, pricing, and availability are set up
4. **Review**: Optional administrative review for platform standards
5. **Publication**: Event becomes visible to the public
6. **Active**: During the event timeframe
7. **Completion**: After event date, enters historical state
8. **Archival**: Long-term storage for completed events

AWS Integration

The Event Module integrates with several AWS services for scalability and performance:

S3 Integration

```
// Event image storage and processing
@Injectable()
export class EventImageService {
  constructor(
    @Inject(S3_CLIENT_TOKEN) private readonly s3Client: S3Client,
    private readonly configService: ConfigService,
  ) {}

  async uploadEventImage(
    eventId: string,
    imageType: 'banner' | 'featured',
    file: Express.Multer.File,
  ): Promise<string> {
    const key =
`events/${eventId}/${imageType}/${uuidv4()}.${this.getExtension(file)}`;
    const command = new PutObjectCommand({
      Bucket: this.configService.get('AWS_S3_BUCKET'),
      Key: key,
      Body: file.buffer,
      ContentType: file.mimetype,
      ACL: 'public-read',
    });

    await this.s3Client.send(command);
    return this.generateImageUrl(key);
  }
}
```

CloudFront Integration

- Content delivery network for event images and media
- Edge caching for high-performance global access
- Signed URLs for protected content

SQS Integration

- Asynchronous processing of event-related tasks
- Decoupling of event creation and image processing
- Retry mechanisms for failed operations

Lambda Integration

- Serverless image processing and resizing
- Event thumbnail generation
- Automated SEO metadata extraction

Module Interactions

Event & User Module

- Event organizers are authenticated users with specific roles
- User preferences influence event recommendations
- User activity history shapes event discovery

Event & Ticket Module

- Events define the available ticket inventory
- Ticket availability affects event visibility and status
- Event changes may trigger ticket status updates

Event & Image Module

- Events require media assets for display
- Image processing pipelines optimize event visual content
- CDN integration ensures fast global delivery

Event & Tag Module

- Events are categorized using hierarchical tags
- Tags power the discovery and recommendation system
- Tag analytics inform content strategy

Event & Search Module

- Events are indexed for full-text and faceted search
- Geolocation-based event discovery
- Personalized search results based on user preferences

Event & Notification Module

- Event status changes trigger notifications
- Upcoming event reminders for interested users
- Real-time alerts for event modifications

Advanced Features

Geo-Location Services

- Events indexed by geographic coordinates
- Radius-based search functionality
- Integration with mapping services for venue visualization
- Location-based recommendations

SEO Optimization

- Automated generation of search-friendly URLs
- Structured data markup for event schema

- Dynamic sitemap generation for improved discoverability
- Meta tag optimization for social sharing

Analytics Integration

- Real-time tracking of event page views
- Conversion funnels for ticket purchase analysis
- A/B testing framework for event presentation
- Heat mapping of user interactions

Content Management

- Rich text editor for event descriptions
- Media gallery management for event assets
- Template system for consistent event presentation
- Version control for event content changes

Scalability Considerations

1. **Read/Write Separation**
    - Read-heavy operations utilize caching and read replicas
    - Write operations are carefully optimized and rate-limited
    - Eventual consistency model for high-volume scenarios
2. **Seasonal Scaling**
    - Auto-scaling based on anticipated event traffic patterns
    - Predictive scaling for known high-volume periods
    - Resource allocation based on event popularity metrics
3. **Global Distribution**
    - Multi-region deployment for global audience
    - Database sharding by geographic region
    - Localized content delivery through CDN edge locations

Event Management API

```
@Controller('events')
export class EventController {
  constructor(private readonly eventService: EventService) {}

  @Post()
  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(UserRole.ORGANIZER, UserRole.ADMIN)
  async createEvent(@Body() createEventDto: CreateEventDto, @User() user: UserEntity): Promise<EventEntity> {
    return this.eventService.createEvent(createEventDto, user.id);
  }
  @Get(':id')
  async getEvent(@Param('id') id: string): Promise<EventEntity> {
    return this.eventService.findEventById(id);
  }

  @Patch(':id')
  @UseGuards(JwtAuthGuard, EventOwnerGuard)
  async updateEvent(
    @Param('id') id: string,
    @Body() updateEventDto: UpdateEventDto,
  ): Promise<EventEntity> {
    return this.eventService.updateEvent(id, updateEventDto);
  }

  @Post(':id/publish')
  @UseGuards(JwtAuthGuard, EventOwnerGuard)
  async publishEvent(@Param('id') id: string): Promise<EventEntity> {
    return this.eventService.publishEvent(id);
  }

  @Delete(':id')
  @UseGuards(JwtAuthGuard, EventOwnerGuard)
  async cancelEvent(@Param('id') id: string, @Body() cancelEventDto: CancelEventDto): Promise<EventEntity> {
    return this.eventService.cancelEvent(id, cancelEventDto.reason);
  }
}
```

Ticket Analytics & Reporting

The platform provides comprehensive analytics and reporting capabilities for ticket sales and attendance data:

Real-time Dashboards

- **Sales Velocity**: Tickets sold per minute/hour/day
- **Inventory Status**: Current availability by ticket class
- **Revenue Metrics**: Gross and net revenue with tax breakdown
- **Conversion Funnels**: From page view to completed purchase
- **Geographic Distribution**: Buyer location analytics

Organizer Reports

- **Sales Summary**: Overview of ticket sales and revenue
- **Attendance Tracking**: Real-time and historical check-in rates
- **Demographic Analysis**: Anonymized attendee demographics
- **Purchase Patterns**: Time-of-day and day-of-week analytics
- **Refund/Cancellation Analysis**: Patterns and financial impact

Predictive Analytics

- **Sales Projections**: ML-based forecasting of sellout timing
- **Price Optimization**: Suggested price points based on demand
- **Attendance Predictions**: Expected show rates and no-shows
- **Upsell Opportunities**: Identification of potential package upgrades
- **Fraud Risk Scoring**: Identification of suspicious purchasing patterns

Export Capabilities

- **CSV/Excel**: Tabular data for external analysis
- **PDF Reports**: Formatted reports for stakeholders
- **API Access**: Programmatic access to analytics data
- **Scheduled Reports**: Automated delivery to stakeholders
- **Custom Queries**: Flexible report building for specific needs

Testing Strategy

The backend implements a comprehensive testing strategy:

Test Types

1. **Unit Tests**: For isolated service and controller functionality
2. **Integration Tests**: For testing module interactions
3. **E2E Tests**: For complete user workflows

Testing Approach

- **TDD/BDD**: Tests are written before or alongside implementation
- **Mocking**: External dependencies are mocked for unit testing
- **Continuous Testing**: Tests run on each code change
- **Test Coverage**: Targeting high coverage for critical paths

Error Handling

The system implements a robust error handling strategy:

Error Types

- **Validation Errors**: For invalid input data
- **Authentication Errors**: For security-related issues
- **Not Found Errors**: For missing resources
- **Conflict Errors**: For business rule violations
- **Internal Errors**: For unexpected system issues

Error Response Format

```json
{
  "statusCode": 400,
  "message": "Error description",
  "error": "Error type",
  "details": { "field": "Specific error reason" }
}
```

Logging

- Structured logging with context for easier debugging
- Error levels (DEBUG, INFO, WARN, ERROR)
- Request/response logging for API calls
- Performance metrics logging

Deployment

The application supports multiple deployment strategies:

Development Environment

- Local development with Docker Compose
- Hot-reloading for faster development

Production Environment

- Containerized deployment with Docker
- Horizontal scaling capabilities
- Environment-specific configuration via environment variables

CI/CD Pipeline

- Automated testing and deployment
- Infrastructure as Code principles
- Rolling updates for zero-downtime deployments

Future Improvements

Planned enhancements for the backend:

1. **Microservices Architecture**: Evolution toward domain-specific services
2. **GraphQL API**: Additional API layer for complex data requirements
3. **Real-time Features**: WebSocket integration for live updates on ticket availability and event changes
4. **Analytics**: Enhanced reporting and metrics collection for event performance and ticket sales
5. **Multi-tenancy**: Support for multiple event organizers with isolated data
6. **Dynamic Pricing Engine**: Advanced algorithms for demand-based ticket pricing
7. **Blockchain Ticketing**: Explore blockchain for ticket authenticity and resale control
8. **AI-Powered Fraud Detection**: Machine learning models to identify suspicious ticket activities
9. **Enhanced Waitlist System**: Sophisticated queuing mechanism for high-demand events
10. **International Payment Support**: Expand payment options for global market reach

Getting Started

Prerequisites

- Node.js 18+
- MongoDB
- Redis
- AWS Account (for S3 storage)
- PayOS API credentials

Installation

```
# Install dependencies
$ npm install

# Setup environment variables
$ cp .env.example .env

# Generate Prisma client
$ npx prisma generate

# Run database migrations
$ npx prisma db push

# Start development server
$ npm run start:dev
```

Environment Variables

Key environment variables required:

- `DATABASE_URL`: MongoDB connection string
- `REDIS_URL`: Redis connection URL
- `JWT_SECRET`: Secret key for JWT signing
- `PAYOS_CLIENT_ID`: PayOS client ID
- `PAYOS_API_KEY`: PayOS API key
- `AWS_ACCESS_KEY`: AWS access key for S3
- `AWS_SECRET_KEY`: AWS secret key for S3

License

This project is licensed under the MIT License - see the LICENSE file for details.