Software Requirements Specification

Event Platform
Version 2.0 - Comprehensive

Event Development Team

December 16, 2025

Abstract

This comprehensive Software Requirements Specification (SRS) document provides an in-depth description of the Event platform, a full-stack events and ticketing system with real-time social features. The document follows IEEE 830-1998 standard and includes detailed functional requirements, non-functional attributes, implementation specifications, error handling mechanisms, data models, system architecture, consequences of requirement failures, and traceability matrices. This version includes extensive implementation details, technical specifications, and operational consequences based on the completed system.

Contents

1 Introduction

1.1 Purpose

This Software Requirements Specification (SRS) document serves as a comprehensive specification for the Event platform. The document:

- Provides detailed functional and non-functional requirements
- Serves as a contract between stakeholders and development team
- Guides design, development, testing, and maintenance phases
- Establishes baseline for system validation and verification
- Documents implementation details and technical decisions
- Specifies error handling and failure consequences
- Provides traceability between requirements and implementation

1.2 Scope

Event is a comprehensive event management and ticketing platform combining:

- **Event Discovery**: Search, filter, and browse events
- **Ticket Booking**: Lock, confirm, and cancel ticket reservations
- **Real-Time Social**: WebSocket-based chat during events
- **Notifications**: Email/SMS and webhook-based event notifications
- **Analytics**: Batch processing of revenue, user activity, and event metrics
- **Media Management**: Image upload and CDN-like distribution
- **Feed System**: Personalized event recommendations

System Boundaries:

- **In Scope**: All microservices, frontend applications, infrastructure components, APIs, data persistence, caching, message queuing
- **Out of Scope**: Payment gateway integration (mock implementation), SMS/Email provider integration (logging only), third-party authentication (OAuth), mobile native applications

1.3 Overview

This document is organized as follows:

- Section 2: General Description
- Section 3: System Architecture
- Section 4: Functional Requirements (with implementation details and consequences)
- Section 5: Data Models and Relationships
- Section 6: Interface Requirements
- Section 7: Performance Requirements
- Section 8: Design Constraints
- Section 9: Non-Functional Attributes
- Section 10: Error Handling and Exception Management

- Section 11: Security Requirements
- Section 12: Deployment and Operations
- Section 13: Appendices

2 General Description

2.1 Product Perspective

Project operates as a standalone, web-based microservices platform. The system integrates with:

External Systems:

- **Email/SMS Providers**: Via webhook callbacks (mock implementation logs notifications)
- **Payment Gateways**: Future integration (currently mock payment processing)
- **CDN Services**: Simulated via Nginx with caching headers
- **Monitoring Tools**: Spring Actuator endpoints (future: Prometheus/Grafana)

Infrastructure Components:

- **PostgreSQL 16**: Primary relational database with separate schemas per service
- **Redis 7**: Caching, rate limiting, session storage, presence tracking, message storage
- **Kafka 7.6.0**: Asynchronous message queue for event-driven architecture
- **Elasticsearch**: Full-text search engine for event discovery
- **Nginx**: Reverse proxy, load balancer, and static file server
- **Docker**: Containerization platform for all services

Client Applications:

- Modern web browsers (Chrome, Firefox, Safari, Edge) with JavaScript enabled
- WebSocket support required for real-time chat features

2.2 Product Functions

The system provides eight major functional areas:

1. **Authentication & Authorization**: User registration, login, JWT-based session management, role-based access control
2. **Event Management**: CRUD operations, search, filtering, categorization, popularity scoring
3. **Ticket Booking**: Seat locking, confirmation with idempotency, cancellation, availability checking
4. **Real-Time Communication**: WebSocket chat, presence tracking, read receipts, typing indicators
5. **Notifications**: Kafka-based event consumption, email/SMS notifications, webhook delivery with retry logic
6. **Media Management**: Image upload, validation, storage, CDN-like URL generation
7. **Analytics**: Batch processing of revenue metrics, daily active users, top events, dashboard summaries
8. **Feed & Recommendations**: Personalized event feeds with Redis-backed caching

2.3 User Characteristics

The system serves three primary user types:

End Users:

- General public seeking to discover and attend events
- Expected to have basic web browsing skills
- May access from desktop or mobile devices
- No technical expertise required

Event Organizers:

- Users creating and managing events
- Expected to understand event management concepts
- May manage multiple events simultaneously
- Require access to analytics and management tools

Administrators:

- System administrators managing platform operations
- Technical expertise in system administration
- Access to all system features and administrative functions

2.4 Operating Environment

Server Environment:

- Linux-based containers (Docker) running Spring Boot 3.x services
- Java 17+ runtime environment
- Minimum 4GB RAM per service instance
- Minimum 2 CPU cores per service instance
- Network latency: Maximum 100ms between services

Client Environment:

- Modern web browsers with JavaScript enabled
- WebSocket protocol support (RFC 6455)
- HTTPS/TLS 1.2+ support
- Minimum screen resolution: 320x568 (mobile), 1024x768 (desktop)

Development Environment:

- Docker Desktop or Docker Engine
- docker-compose for local orchestration
- Node.js 18+ for frontend development
- Gradle 7+ for backend builds

2.5 Design and Implementation Constraints

- **Architecture**: Microservices pattern with Spring Boot framework
- **API Design**: RESTful APIs only (no GraphQL/gRPC)
- **Authentication**: JWT-based stateless authentication
- **Deployment**: Docker containerization required
- **Database**: PostgreSQL for relational data, Redis for caching
- **Message Queue**: Kafka for asynchronous event processing
- **Search**: Elasticsearch for full-text search capabilities
- **State Management**: Services must be stateless (except WebSocket connections)

2.6 Assumptions and Dependencies

Assumptions:

- Users have stable internet connectivity
- External notification services are available (currently mocked)
- Infrastructure components are operational and monitored
- WebSocket connections are supported by client browsers
- JWT tokens are securely stored on client side
- Database backups are performed regularly (future requirement)
- Network security is handled at infrastructure level

Dependencies:

- PostgreSQL database availability
- Redis cache availability
- Kafka message broker availability
- Elasticsearch cluster availability
- Nginx reverse proxy availability
- Docker runtime availability

3 System Architecture

3.1 Microservices Architecture

Project follows a microservices architecture pattern with eight independent services:

1. **api-gateway**: Spring Cloud Gateway routing all external requests
2. **auth-service**: User authentication and authorization
3. **event-service**: Event CRUD, search, feed generation
4. **ticket-service**: Ticket booking, locking, confirmation
5. **notification-service**: Event notifications and webhook delivery
6. **chat-service**: Real-time messaging via WebSocket
7. **media-service**: Image upload and storage
8. **analytics-service**: Batch analytics processing

3.2 Communication Patterns

- **Synchronous**: REST API calls between frontend and backend via API Gateway
- **Asynchronous**: Kafka message queue for ticket events → notifications
- **Real-Time**: WebSocket connections for chat functionality
- **Service-to-Service**: HTTP calls within Docker network

3.3 Data Flow

1. Client requests routed through Nginx → API Gateway
2. API Gateway validates JWT, applies rate limiting, routes to service
3. Service processes request, queries database/cache
4. Ticket confirmations publish events to Kafka
5. Notification service consumes events, sends notifications
6. Analytics service processes events for batch analytics

4 Functional Requirements

4.1 Authentication Service (FR-AUTH)

4.1.1 FR-AUTH-1: User Registration

Priority: High
Description: The system shall allow new users to register with email and password.

Inputs:

- Email address (string, required, valid email format, max 255 chars)
- Password (string, required, minimum 8 characters, max 128 chars)
- Optional: User role (enum: USER, ORGANIZER, ADMIN, default: USER)

Processing:

1. Validate email format using regex pattern
2. Check email uniqueness in database (unique constraint)
3. Validate password strength (minimum 8 characters)
4. Hash password using BCrypt with salt rounds = 10
5. Create User entity with email, hashed password, role, timestamps
6. Persist to PostgreSQL database (ACID transaction)
7. Generate JWT token with claims: userId, email, roles, expiration (24 hours)
8. Return token to client

Outputs:

- HTTP 201 Created with JSON: {"token": "jwt-token-string", "user": {"id": 1, "email": "user@example.com"}}
- HTTP 400 Bad Request if validation fails: {"message": "field error message"}
- HTTP 409 Conflict if email exists: {"message": "Email already registered"}

Implementation Details:

- Uses Spring Data JPA for database operations
- BCryptPasswordEncoder for password hashing
- JWT token signed with HMAC-SHA256 algorithm
- Email uniqueness enforced via database unique constraint
- Transaction rollback on any failure

Consequences of Failure:

- **Email Already Exists**: User cannot register, must use login or password reset
- **Weak Password**: Security vulnerability, potential account compromise
- **Database Failure**: Registration fails, user sees error message, must retry
- **JWT Generation Failure**: User registered but cannot login immediately
- **Transaction Rollback**: No partial user creation, data consistency maintained

JML Specifications:

```
/*@ public normal_behavior
  @ requires request != null;
  @ requires request.getEmail() != null && request.getEmail().length() > 0 && request.getEmail().length() <= 255;
  @ requires request.getEmail().matches("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$");
  @ requires request.getPassword() != null && request.getPassword().length() >= 8 && request.getPassword().length() <= 128;
  @ requires !userRepository.existsByEmail(request.getEmail());
  @ ensures \result != null;
  @ ensures \result.getToken() != null && \result.getToken().length() > 0;
  @ ensures \result.getId() > 0;
  @ ensures userRepository.existsById(\result.getId());
  @ ensures userRepository.findByEmail(request.getEmail()).isPresent();
  @ ensures userRepository.findByEmail(request.getEmail()).get().getPasswordHash() != request.getPassword();
  @ ensures passwordEncoder.matches(request.getPassword(), userRepository.findByEmail(request.getEmail()).get().getPasswordHash());
  @ ensures jwtService.isValid(\result.getToken());
  @ also
  @ public exceptional_behavior
  @ requires request != null && request.getEmail() != null && userRepository.existsByEmail(request.getEmail());
  @ signals (ResponseStatusException) true;
  @ also
  @ public exceptional_behavior
  @ requires request != null && (request.getPassword() == null || request.getPassword().length() < 8);
  @ signals (IllegalArgumentException) true;
  @*/
public AuthResponse register(RegisterRequest request);
```

4.1.2 FR-AUTH-2: User Login

Priority: High
Description: The system shall authenticate users and issue JWT tokens.

Inputs:

- Email address (string, required)
- Password (string, required)

Processing:

1. Extract client IP address for rate limiting
2. Check Redis for rate limit counter (key: "ip:{ip}")
3. If rate limit exceeded, reject request (429 Too Many Requests)
4. Query database for user by email
5. If user not found, increment failed login counter, return 401
6. Verify password hash using BCryptPasswordEncoder.matches()
7. If password invalid, increment failed login counter, return 401
8. Generate JWT token with user claims

9. Store rate limit counter in Redis (TTL: 1 minute)
10. Return token to client

Outputs:

- HTTP 200 OK with JSON: {"token": "jwt-token-string"}
- HTTP 401 Unauthorized: {"message": "Invalid email or password"}
- HTTP 429 Too Many Requests: {"message": "Rate limit exceeded"}

Implementation Details:

- Rate limiting: 5 requests per minute per IP address
- Failed login attempts logged but not locked (future: account lockout)
- JWT expiration: 24 hours from issue time
- Token stored in Authorization header: "Bearer {token}"

Consequences of Failure:

- **Invalid Credentials**: User cannot access system, must retry or reset password
- **Rate Limit Exceeded**: Prevents brute force attacks, user must wait
- **Database Unavailable**: Login fails, system returns 503 Service Unavailable
- **Redis Unavailable**: Rate limiting disabled, login proceeds (graceful degradation)
- **JWT Generation Failure**: Login fails, user must retry

JML Specifications:

```
/*@ public normal_behavior
  @ requires request != null;
  @ requires request.getEmail() != null && request.getEmail().length() > 0;
  @ requires request.getPassword() != null && request.getPassword().length() > 0;
  @ requires userRepository.findByEmail(request.getEmail()).isPresent();
  @ requires passwordEncoder.matches(request.getPassword(),
  @          userRepository.findByEmail(request.getEmail()).get().getPasswordHash());
  @ ensures \result != null;
  @ ensures \result.getToken() != null && \result.getToken().length() > 0;
  @ ensures \result.getId() > 0;
  @ ensures jwtService.isValid(\result.getToken());
  @ ensures jwtService.getSubject(\result.getToken()).equals(request.getEmail());
  @ also
  @ public exceptional_behavior
  @ requires request != null && !userRepository.findByEmail(request.getEmail()).isPresent();
  @ signals (ResponseStatusException) true;
  @ also
  @ public exceptional_behavior
  @ requires request != null && userRepository.findByEmail(request.getEmail()).isPresent() &&
  @          !passwordEncoder.matches(request.getPassword(),
  @          userRepository.findByEmail(request.getEmail()).get().getPasswordHash());
  @ signals (ResponseStatusException) true;
  @*/
public AuthResponse login(LoginRequest request);
```

4.1.3 FR-AUTH-3: Get Current User

Priority: Medium
Description: The system shall return authenticated user information.
Inputs: JWT token in Authorization header
Processing:

1. Extract JWT from Authorization header
2. Validate token signature using shared secret
3. Check token expiration
4. Extract user ID from token claims
5. Query database for user by ID

6. Return user information (excluding password hash)

Outputs:

- HTTP 200 OK: {"id": 1, "email": "user@example.com", "roles": ["USER"]}
- HTTP 401 Unauthorized if token invalid/expired
- HTTP 404 Not Found if user deleted

Consequences of Failure:

- **Invalid Token**: User must re-authenticate
- **Expired Token**: User must login again
- **User Deleted**: Token valid but user removed, return 404

JML Specifications:

```
/*@ public normal_behavior
  @ requires authHeader != null && authHeader.startsWith("Bearer ");
  @ requires jwtService.isValid(authHeader.substring(7));
  @ requires userRepository.findByEmail(jwtService.getSubject(authHeader.substring(7))).isPresent();
  @ ensures \result != null;
  @ ensures \result.getId() > 0;
  @ ensures \result.getEmail() != null;
  @ ensures \result.getEmail().equals(jwtService.getSubject(authHeader.substring(7)));
  @ ensures \result.getCreatedAt() != null;
  @ also
  @ public exceptional_behavior
  @ requires authHeader == null || !authHeader.startsWith("Bearer ");
  @ signals (ResponseStatusException) true;
  @ also
  @ public exceptional_behavior
  @ requires authHeader != null && authHeader.startsWith("Bearer ") &&
  @          !jwtService.isValid(authHeader.substring(7));
  @ signals (ResponseStatusException) true;
  @ also
  @ public exceptional_behavior
  @ requires authHeader != null && authHeader.startsWith("Bearer ") &&
  @          jwtService.isValid(authHeader.substring(7)) &&
  @          !userRepository.findByEmail(jwtService.getSubject(authHeader.substring(7))).isPresent();
  @ signals (ResponseStatusException) true;
  @*/
public UserProfileResponse getCurrentUser(String authHeader);
```

4.2 Event Service (FR-EVENT)

4.2.1 FR-EVENT-1: Create Event

Priority: High
Description: Authenticated organizers shall create events.

Inputs:

- Title (string, max 140 characters, required)
- Description (text, optional, max 10,000 chars)
- City (string, max 64 characters, required)
- Event time (OffsetDateTime, ISO-8601 format, required, future date)
- Capacity (integer, required, > 0, $< 1{,}000{,}000$)
- Venue (string, optional, max 255 chars)
- Category (string, optional, max 64 chars)
- Price (BigDecimal, precision 12, scale 2, default: 0, $>= 0$)
- Image URL (string, optional, max 512 chars, valid URL format)
- Public event flag (boolean, default: true)
- Organizer ID (Long, extracted from JWT, not in request body)

Processing:

1. Validate organizer permissions from JWT (must have ORGANIZER or ADMIN role)
2. Validate all input fields (Bean Validation annotations)
3. Resolve shard ID by city using CityBasedShardResolver
4. Set ShardContext thread-local variable
5. Map DTO to Entity using EventMapper
6. Set default price to 0 if null
7. Set default popularityScore to 0
8. Initialize version field to 0 (optimistic locking)
9. Persist event to PostgreSQL (ACID transaction)
10. Sync seat inventory with ticket-service via HTTP call
11. Reconcile capacity if ticket-service returns different value
12. Add event ID to Bloom filter for fast existence checks
13. Index event in Elasticsearch (async, errors ignored)
14. Evict popularEvents cache (all entries)
15. Clear ShardContext
16. Return EventResponse DTO

Outputs:

- HTTP 201 Created with EventResponse JSON
- HTTP 400 Bad Request if validation fails
- HTTP 403 Forbidden if user not organizer
- HTTP 500 Internal Server Error if database/Elasticsearch fails

Implementation Details:

- Shard resolution: City-based hashing (e.g., "New York" $\rightarrow$ ShardId.SHARD_1)
- Bloom filter: Custom implementation, false positive rate $< 1\%$
- Elasticsearch indexing: Non-blocking, failures logged but don't fail request
- Cache eviction: Spring Cache @CacheEvict annotation
- Transaction: @Transactional ensures atomicity

Consequences of Failure:

- **Validation Failure**: Event not created, user sees specific field errors
- **Unauthorized Access**: Security breach prevented, 403 returned
- **Database Failure**: Transaction rolls back, no partial event creation
- **Ticket Service Unavailable**: Event created but seat inventory not synced, must retry
- **Elasticsearch Failure**: Event created but not searchable until re-indexed
- **Cache Eviction Failure**: Stale popular events may be shown temporarily

JML Specifications:

```
/*@ public normal_behavior
  @ requires req != null;
  @ requires req.getTitle() != null && req.getTitle().length() > 0 && req.getTitle().length() <= 140;
  @ requires req.getCity() != null && req.getCity().length() > 0 && req.getCity().length() <= 64;
  @ requires req.getTime() != null && req.getTime().isAfter(OffsetDateTime.now());
  @ requires req.getCapacity() > 0 && req.getCapacity() < 1000000;
  @ requires req.getPrice() == null || (req.getPrice().compareTo(BigDecimal.ZERO) >= 0);
  @ ensures \result != null;
  @ ensures \result.getId() > 0;
  @ ensures \result.getTitle().equals(req.getTitle());
  @ ensures \result.getCity().equals(req.getCity());
  @ ensures \result.getCapacity() == req.getCapacity();
  @ ensures eventRepository.existsById(\result.getId());
  @ ensures bloomFilter.mightContain(String.valueOf(\result.getId()));
  @ ensures \result.getPrice() != null && \result.getPrice().compareTo(BigDecimal.ZERO) >= 0;
  @ also
  @ public exceptional_behavior
  @ requires req != null && (req.getTitle() == null || req.getTitle().length() == 0 ||
  @          req.getTitle().length() > 140);
  @ signals (IllegalArgumentException) true;
  @ also
  @ public exceptional_behavior
  @ requires req != null && (req.getCapacity() <= 0 || req.getCapacity() >= 1000000);
  @ signals (IllegalArgumentException) true;
  @*/
public EventResponse createEvent(EventCreateRequest req);
```

4.2.2 FR-EVENT-2: List Events

Priority: High
Description: Users shall retrieve paginated list of events with filtering.

Inputs:

- City (string, optional)
- Organizer ID (Long, optional)
- From time (OffsetDateTime, ISO-8601, optional)
- To time (OffsetDateTime, ISO-8601, optional)
- Sort order (enum: TIME_ASC, TIME_DESC, POPULAR, default: TIME_ASC)
- Page number (integer, default: 0, >= 0)
- Page size (integer, default: 20, min: 1, max: 100)

Processing:

1. Build Specification object from filter parameters
2. If sort = POPULAR, check Redis cache (key: "popularEvents:{city}:{page}")
3. If cache hit, return cached results
4. If cache miss or sort != POPULAR, query database with Specification
5. Apply pagination using Spring Data Pageable
6. Use database indexes: idx_events_city_time, idx_events_organizer_id
7. If POPULAR sort, compute popularity scores, cache results (TTL: 10 minutes)
8. Return Page object with events and pagination metadata

Outputs:

- HTTP 200 OK with paginated response:

```
{
  "content": [/* EventResponse objects */],
  "totalElements": 150,
  "totalPages": 8,
  "number": 0,
  "size": 20,
  "first": true,
  "last": false
}
```

Implementation Details:

- Database indexes optimize city + time range queries
- Cache key includes city and page for cache locality
- Popularity score computed from popularityScore field
- Pagination prevents large result sets

Consequences of Failure:

- **Cache Miss**: Slightly slower response, but correct results returned
- **Database Slow Query**: Response time increases, may timeout
- **Invalid Pagination**: Returns empty page if page number too high
- **Index Missing**: Query performance degrades significantly

JML Specifications:

```
/*@ public normal_behavior
  @ requires filter != null;
  @ requires filter.getPage() >= 0;
  @ requires filter.getSize() > 0 && filter.getSize() <= 100;
  @ ensures \result != null;
  @ ensures \result.getContent() != null;
  @ ensures \result.getTotalElements() >= 0;
  @ ensures \result.getTotalPages() >= 0;
  @ ensures \result.getNumber() == filter.getPage();
  @ ensures \result.getSize() == filter.getSize();
  @ ensures \result.getContent().size() <= filter.getSize();
  @ ensures filter.getCity() == null || filter.getCity().isBlank() ||
  @          (\forall int i; 0 <= i && i < \result.getContent().size();
  @           \result.getContent().get(i).getCity().equalsIgnoreCase(filter.getCity()));
  @ ensures filter.getOrganizerId() == null || (\forall int i; 0 <= i && i < \result.getContent().size();
  @          \result.getContent().get(i).getOrganizerId().equals(filter.getOrganizerId()));
  @ ensures filter.getFromTime() == null || (\forall int i; 0 <= i && i < \result.getContent().size();
  @          \result.getContent().get(i).getTime().isAfter(filter.getFromTime()) ||
  @          \result.getContent().get(i).getTime().equals(filter.getFromTime()));
  @ ensures filter.getToTime() == null || (\forall int i; 0 <= i && i < \result.getContent().size();
  @          \result.getContent().get(i).getTime().isBefore(filter.getToTime()) ||
  @          \result.getContent().get(i).getTime().equals(filter.getToTime()));
  @*/
public Page<EventResponse> searchEvents(EventFilterRequest filter);
```

4.2.3 FR-EVENT-3: Get Event Details

Priority: High
Description: Users shall retrieve detailed information about a specific event.
Inputs: Event ID (Long, path parameter)
Processing:

1. Check Bloom filter: bloomFilter.mightContain(String.valueOf(id))
2. If Bloom filter returns false, return 404 immediately (avoid DB query)
3. Check Redis cache: cacheManager.getCache("eventDetails").get(id)
4. If cache hit, deserialize EventResponse, return it
5. If cache miss, query database by ID
6. If not found, return 404
7. Map Entity to EventResponse DTO
8. Cache result in Redis (TTL: 600 seconds)
9. Update Bloom filter: bloomFilter.add(String.valueOf(id))
10. Return EventResponse

Outputs:

- HTTP 200 OK with EventResponse JSON

- HTTP 404 Not Found if event doesn't exist

Implementation Details:

- Bloom filter false positive rate: $< 1\%$
- Cache TTL: 10 minutes (600 seconds)
- Cache key: event ID (Long)
- Cache serialization: JSON via RedisTemplate

Consequences of Failure:

- **Bloom Filter False Positive**: Unnecessary DB query, but correct result returned
- **Cache Deserialization Error**: Cache entry evicted, DB query performed
- **Cache Miss**: Slower response (DB query), but correct
- **Database Failure**: 500 error, user cannot view event

JML Specifications:

```
/*@ public normal_behavior
  @ requires eventId > 0;
  @ requires bloomFilter.mightContain(String.valueOf(eventId));
  @ requires eventRepository.existsById(eventId);
  @ ensures \result != null;
  @ ensures \result.getId() == eventId;
  @ ensures \result.getTitle() != null;
  @ ensures \result.getCity() != null;
  @ ensures \result.getCapacity() > 0;
  @ ensures eventRepository.findById(eventId).equals(\result);
  @ also
  @ public exceptional_behavior
  @ requires eventId > 0 && !bloomFilter.mightContain(String.valueOf(eventId));
  @ signals (EntityNotFoundException) true;
  @ also
  @ public exceptional_behavior
  @ requires eventId > 0 && bloomFilter.mightContain(String.valueOf(eventId)) &&
  @          !eventRepository.existsById(eventId);
  @ signals (EntityNotFoundException) true;
  @*/
public EventResponse getEventById(Long eventId);
```

4.2.4 FR-EVENT-4: Search Events

Priority: Medium
Description: Users shall perform full-text search on events.

Inputs:

- Query string (string, required, min: 1 char, max: 200 chars)
- City (string, optional)
- Page number (integer, default: 0)
- Page size (integer, default: 20)

Processing:

1. Build Elasticsearch query with query string
2. Apply city filter if provided
3. Execute search query against Elasticsearch index
4. Retrieve event IDs from search results
5. Query PostgreSQL for full event details (IN clause with IDs)
6. Apply pagination
7. Return paginated results

Outputs: HTTP 200 OK with paginated search results
Consequences of Failure:

- **Elasticsearch Unavailable**: Search fails, return 503, suggest using filter endpoint
- **Empty Query**: Return empty results
- **Index Out of Sync**: Stale results, events may be missing

JML Specifications:

```
/*@ public normal_behavior
  @ requires query != null && query.length() > 0 && query.length() <= 200;
  @ requires page >= 0;
  @ requires size > 0 && size <= 100;
  @ ensures \result != null;
  @ ensures \result.getContent() != null;
  @ ensures \result.getTotalElements() >= 0;
  @ ensures \result.getNumber() == page;
  @ ensures \result.getSize() == size;
  @ ensures \result.getContent().size() <= size;
  @ ensures city == null || city.isBlank() || (\forall int i; 0 <= i && i < \result.getContent().size();
  @          \result.getContent().get(i).getCity().equalsIgnoreCase(city));
  @*/
public Page<EventResponse> searchFullText(String query, String city, int page, int size);
```

4.2.5 FR-EVENT-5: Update Event

Priority: Medium
Description: Event organizers shall update their events.
Inputs: Event ID, updated event data, organizer ID from JWT
Processing:

1. Verify organizer ownership (event.organizerId == JWT userId)
2. Load event with optimistic lock (version field)
3. Validate updated data
4. Update event fields
5. Save with version check (OptimisticLockException if concurrent update)
6. Evict caches: eventDetails, popularEvents
7. Update Elasticsearch index
8. Update Bloom filter

Consequences of Failure:

- **Unauthorized Update**: 403 Forbidden, security maintained
- **Optimistic Lock Conflict**: 409 Conflict, user must refresh and retry
- **Concurrent Updates**: Last write wins, but version prevents lost updates

JML Specifications:

```
/*@ public normal_behavior
  @ requires id > 0;
  @ requires userId > 0;
  @ requires req != null;
  @ requires eventRepository.existsById(id);
  @ requires eventRepository.findById(id).getOrganizerId().equals(userId);
  @ ensures \result != null;
  @ ensures \result.getId() == id;
  @ ensures eventRepository.existsById(id);
  @ ensures req.getTitle() == null || \result.getTitle().equals(req.getTitle());
  @ ensures bloomFilter.mightContain(String.valueOf(id));
  @ also
  @ public exceptional_behavior
  @ requires id > 0 && userId > 0 && eventRepository.existsById(id) &&
  @          !eventRepository.findById(id).getOrganizerId().equals(userId);
  @ signals (IllegalStateException) true;
  @ also
  @ public exceptional_behavior
  @ requires id > 0 && !eventRepository.existsById(id);
  @ signals (EntityNotFoundException) true;
  @*/
public EventResponse updateEvent(Long id, EventCreateRequest req, Long userId);
```

4.2.6 FR-EVENT-6: Delete Event

Priority: Medium
Description: Event organizers shall delete their events.
Inputs: Event ID, organizer ID from JWT
Processing:

1. Verify organizer ownership
2. Check for existing tickets (if tickets exist, prevent deletion or soft delete)
3. Delete event from database
4. Evict all caches
5. Remove from Elasticsearch index
6. Update Bloom filter (optional, false negatives acceptable)

Consequences of Failure:

- **Event Has Tickets**: Cannot delete, must cancel event first
- **Cascade Delete**: Related tickets may be orphaned (prevented by foreign key constraints)

JML Specifications:

```
/*@ public normal_behavior
  @ requires id > 0;
  @ requires userId > 0;
  @ requires eventRepository.existsById(id);
  @ requires eventRepository.findById(id).getOrganizerId().equals(userId);
  @ ensures !eventRepository.existsById(id);
  @ also
  @ public exceptional_behavior
  @ requires id > 0 && userId > 0 && eventRepository.existsById(id) &&
  @          !eventRepository.findById(id).getOrganizerId().equals(userId);
  @ signals (IllegalStateException) true;
  @ also
  @ public exceptional_behavior
  @ requires id > 0 && !eventRepository.existsById(id);
  @ signals (EntityNotFoundException) true;
  @*/
public void deleteEvent(Long id, Long userId);
```

4.2.7 FR-EVENT-7: Get Event Feed

Priority: Medium
Description: Users shall retrieve personalized event feeds.
Inputs:

- Feed type (enum: POPULAR, TRENDING, RECOMMENDED)
- City (string, optional)
- Page number (integer, default: 0)
- Page size (integer, default: 20)
- User ID (from JWT)

Processing:

1. Build Redis key: "feed:{type}:{city}:{userId}"
2. Check Redis Sorted Set (ZSET) for cached feed
3. If cache hit, retrieve top N events by score, return with cached=true flag
4. If cache miss, compute feed:
   (a) Query events from database
   (b) Compute popularity scores (based on popularityScore field)

(c) Sort by score (descending)

(d) Store in Redis ZSET with scores

(e) Set TTL: 10 minutes

5. Return FeedResponse with events and cached flag

Outputs: HTTP 200 OK with FeedResponse containing events and cached flag
Consequences of Failure:

- **Cache Miss**: Slower response, but correct feed generated
- **Redis Unavailable**: Feed computed from database, no caching

JML Specifications:

```
/*@ public normal_behavior
  @ requires feedType != null;
  @ requires userId > 0;
  @ requires page >= 0;
  @ requires size > 0 && size <= 100;
  @ ensures \result != null;
  @ ensures \result.getEvents() != null;
  @ ensures \result.getEvents().size() <= size;
  @ ensures city == null || (\forall int i; 0 <= i && i < \result.getEvents().size();
  @          \result.getEvents().get(i).getCity().equals(city));
  @ ensures feedType == POPULAR ==> (\forall int i; 0 <= i && i < \result.getEvents().size() - 1;
  @          \result.getEvents().get(i).getPopularityScore() >=
  @          \result.getEvents().get(i+1).getPopularityScore());
  @*/
public FeedResponse getFeed(FeedType feedType, String city, Long userId, int page, int size);
```

4.3 Ticket Service (FR-TICKET)

4.3.1 FR-TICKET-1: Lock Tickets

Priority: High
Description: Users shall temporarily lock seats for an event.
Inputs:

- Event ID (Long, required)
- User ID (Long, from JWT)
- Quantity (integer, required, > 0, < 100)

Processing:

1. Load SeatInventory by event ID
2. If inventory not found, throw IllegalArgumentException
3. Check available seats: inventory.getAvailableSeats() >= quantity
4. If insufficient seats, throw IllegalStateException
5. Begin database transaction
6. Decrement available seats: inventory.setAvailableSeats(available - quantity)
7. Save inventory (optimistic lock via version field)
8. Fetch event price from event-service (cached via TtlLruCache)
9. Calculate total price: pricePerSeat $\times$ quantity
10. Create Ticket entity:
    - Status: LOCKED
    - Lock expiration: now + 10 minutes
    - Price: total price
    - Quantity: requested quantity

11. Save ticket to database
12. Commit transaction
13. Return TicketResponse

Outputs:

- HTTP 200 OK with TicketResponse
- HTTP 400 Bad Request if insufficient seats or invalid input
- HTTP 404 Not Found if event doesn't exist

Implementation Details:

- Lock duration: 10 minutes (configurable)
- Optimistic locking prevents concurrent seat overbooking
- Transaction ensures atomic seat decrement and ticket creation
- Price caching reduces cross-service calls

Consequences of Failure:

- **Insufficient Seats**: User cannot lock tickets, must reduce quantity
- **Concurrent Locks**: Optimistic lock prevents overbooking, one request succeeds
- **Transaction Rollback**: No partial state, seats not decremented if ticket creation fails
- **Event Service Unavailable**: Cannot fetch price, lock fails with 503
- **Lock Expires**: User must re-lock if confirmation not completed in 10 minutes

JML Specifications:

```
/*@ public normal_behavior
  @ requires request != null;
  @ requires request.eventId() > 0;
  @ requires request.userId() > 0;
  @ requires request.quantity() > 0 && request.quantity() < 100;
  @ requires seatInventoryRepository.findById(request.eventId()).isPresent();
  @ requires seatInventoryRepository.findById(request.eventId()).get().getAvailableSeats() >= request.quantity();
  @ ensures \result != null;
  @ ensures \result.getId() != null;
  @ ensures \result.getEventId() == request.eventId();
  @ ensures \result.getUserId() == request.userId();
  @ ensures \result.getQuantity() == request.quantity();
  @ ensures \result.getStatus() == TicketStatus.LOCKED;
  @ ensures \result.getLockExpiresAt() != null;
  @ ensures \result.getLockExpiresAt().isAfter(Instant.now());
  @ ensures seatInventoryRepository.findById(request.eventId()).get().getAvailableSeats() ==
  @         \old(seatInventoryRepository.findById(request.eventId()).get().getAvailableSeats()) - request.quantity();
  @ ensures ticketRepository.existsById(\result.getId());
  @ also
  @ public exceptional_behavior
  @ requires request != null && !seatInventoryRepository.findById(request.eventId()).isPresent();
  @ signals (IllegalArgumentException) true;
  @ also
  @ public exceptional_behavior
  @ requires request != null && seatInventoryRepository.findById(request.eventId()).isPresent() &&
  @          seatInventoryRepository.findById(request.eventId()).get().getAvailableSeats() < request.quantity();
  @ signals (IllegalStateException) true;
  @*/
public TicketResponse lockTickets(LockTicketRequest request);
```

4.3.2 FR-TICKET-2: Confirm Ticket

Priority: High
Description: Users shall confirm ticket booking with idempotency protection.

Inputs:

- Ticket ID (UUID, required)
- User ID (Long, from JWT)
- Idempotency Key (string, optional, from header or body, UUID format)

Processing:

1. If idempotency key provided, query database: findByIdempotencyKey(key)
2. If existing ticket found, return it immediately (idempotent response)
3. Load ticket by ID and user ID
4. Verify ticket belongs to user
5. Verify ticket status is LOCKED
6. Verify lock hasn't expired: lockExpiresAt > now
7. Begin transaction
8. Update ticket status to CONFIRMED
9. Set idempotency key if provided
10. Save ticket
11. Publish TICKET_CONFIRMED event to Kafka topic "ticket-events"
12. Commit transaction
13. Return confirmed TicketResponse

Outputs:

- HTTP 200 OK with confirmed TicketResponse
- HTTP 400 Bad Request if ticket invalid/expired
- HTTP 409 Conflict if idempotency key duplicate (but return existing ticket)

Implementation Details:

- Idempotency key stored in database with unique index
- Kafka producer uses async send with callback
- Transaction commits before Kafka send (at-least-once delivery)
- Idempotency prevents duplicate charges

Consequences of Failure:

- **Duplicate Idempotency Key**: Returns existing ticket, prevents double booking
- **Lock Expired**: User must re-lock tickets, previous lock released
- **Kafka Unavailable**: Ticket confirmed but notification not sent, retry mechanism handles
- **Transaction Failure**: Ticket remains LOCKED, user can retry
- **Concurrent Confirmation**: Optimistic lock prevents double confirmation

JML Specifications:

```
/*@ public normal_behavior
  @ requires request != null;
  @ requires request.ticketId() != null;
  @ requires request.userId() > 0;
  @ requires request.idempotencyKey() != null && request.idempotencyKey().length() > 0;
  @ requires ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).isPresent();
  @ requires ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).get().getStatus() == TicketStatus.LOCKED;
  @ requires ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).get().getLockExpiresAt().isAfter(Instant.now());
  @ requires !ticketRepository.findByIdempotencyKey(request.idempotencyKey()).isPresent();
  @ ensures \result != null;
  @ ensures \result.getId().equals(request.ticketId());
  @ ensures \result.getStatus() == TicketStatus.CONFIRMED;
  @ ensures ticketRepository.findById(request.ticketId()).get().getStatus() == TicketStatus.CONFIRMED;
  @ ensures ticketRepository.findById(request.ticketId()).get().getIdempotencyKey().equals(request.idempotencyKey());
  @ also
  @ public normal_behavior
```

```
  @ requires request != null && request.idempotencyKey() != null &&
  @          ticketRepository.findByIdempotencyKey(request.idempotencyKey()).isPresent();
  @ ensures \result != null;
  @ ensures ticketRepository.findByIdempotencyKey(request.idempotencyKey()).get().getIdempotencyKey().equals(request.idempotencyKey());
  @ ensures \result.getStatus() == TicketStatus.CONFIRMED;
  @ also
  @ public exceptional_behavior
  @ requires request != null && !ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).isPresent();
  @ signals (IllegalArgumentException) true;
  @ also
  @ public exceptional_behavior
  @ requires request != null && ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).isPresent() &&
  @          ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).get().getStatus() != TicketStatus.LOCKED;
  @ signals (IllegalStateException) true;
  @ also
  @ public exceptional_behavior
  @ requires request != null && ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).isPresent() &&
  @          ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).get().getLockExpiresAt().isBefore(Instant.now());
  @ signals (IllegalStateException) true;
  @*/
public TicketResponse confirmTicket(ConfirmTicketRequest request);
```

4.3.3 FR-TICKET-3: Cancel Ticket

Priority: Medium
Description: Users shall cancel confirmed tickets.

Inputs:

- Ticket ID (UUID, required)
- User ID (Long, from JWT)

Processing:

1. Load ticket by ID and user ID
2. Verify ticket status is CONFIRMED
3. Begin transaction
4. Update ticket status to CANCELLED
5. Increment available seats in inventory
6. Save ticket and inventory
7. Publish TICKET_CANCELLED event to Kafka
8. Commit transaction

Consequences of Failure:

- **Ticket Already Cancelled**: Idempotent operation, return success
- **Locked Ticket**: Cannot cancel locked ticket, must wait for expiration

JML Specifications:

```
/*@ public normal_behavior
  @ requires request != null;
  @ requires request.ticketId() != null;
  @ requires request.userId() > 0;
  @ requires ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).isPresent();
  @ requires ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).get().getStatus() == TicketStatus.CONFIRMED ||
  @          ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).get().getStatus() == TicketStatus.LOCKED;
  @ ensures \result != null;
  @ ensures \result.getId().equals(request.ticketId());
  @ ensures \result.getStatus() == TicketStatus.CANCELLED;
  @ ensures ticketRepository.findById(request.ticketId()).get().getStatus() == TicketStatus.CANCELLED;
  @ ensures seatInventoryRepository.findById(\result.getEventId()).get().getAvailableSeats() ==
  @          \old(seatInventoryRepository.findById(\result.getEventId()).get().getAvailableSeats()) +
  @          \result.getQuantity();
  @ also
  @ public normal_behavior
  @ requires request != null && ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).isPresent() &&
  @          ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).get().getStatus() == TicketStatus.CANCELLED;
  @ ensures \result != null;
  @ ensures \result.getStatus() == TicketStatus.CANCELLED;
  @ also
  @ public exceptional_behavior
  @ requires request != null && !ticketRepository.findByIdAndUserId(request.ticketId(), request.userId()).isPresent();
  @ signals (IllegalArgumentException) true;
  @*/
public TicketResponse cancelTicket(CancelTicketRequest request);
```

4.3.4 FR-TICKET-4: List User Tickets

Priority: Medium
Description: Users shall retrieve their ticket history.
Inputs: User ID (from JWT)
Processing:

1. Query tickets by user ID: findByUserIdOrderByCreatedAtDesc(userId)
2. Use database index: idx_tickets_user
3. Map entities to TicketResponse DTOs
4. Return list

Outputs: HTTP 200 OK with list of TicketResponse objects
JML Specifications:

```
/*@ public normal_behavior
  @ requires userId > 0;
  @ ensures \result != null;
  @ ensures (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getUserId() == userId);
  @ ensures (\forall int i; 0 <= i && i < \result.size() - 1;
  @          \result.get(i).getCreatedAt().isAfter(\result.get(i+1).getCreatedAt()) ||
  @          \result.get(i).getCreatedAt().equals(\result.get(i+1).getCreatedAt()));
  @*/
public List<TicketResponse> listTicketsForUser(Long userId);
```

4.3.5 FR-TICKET-5: Get Seat Availability

Priority: Medium
Description: Users shall check available seats for an event.
Inputs: Event ID (Long, path parameter)
Processing:

1. Load SeatInventory by event ID
2. Return total seats and available seats

Outputs: HTTP 200 OK with SeatAvailabilityResponse
JML Specifications:

```
/*@ public normal_behavior
  @ requires eventId > 0;
  @ requires seatInventoryRepository.findById(eventId).isPresent();
  @ ensures \result != null;
  @ ensures \result.getEventId() == eventId;
  @ ensures \result.getTotalSeats() > 0;
  @ ensures \result.getAvailableSeats() >= 0;
  @ ensures \result.getAvailableSeats() <= \result.getTotalSeats();
  @ ensures \result.getTotalSeats() == seatInventoryRepository.findById(eventId).get().getTotalSeats();
  @ ensures \result.getAvailableSeats() == seatInventoryRepository.findById(eventId).get().getAvailableSeats();
  @ also
  @ public exceptional_behavior
  @ requires eventId > 0 && !seatInventoryRepository.findById(eventId).isPresent();
  @ signals (IllegalArgumentException) true;
  @*/
public SeatAvailabilityResponse getSeatAvailability(Long eventId);
```

4.4 Chat Service (FR-CHAT)

4.4.1 FR-CHAT-1: Send Message via WebSocket

Priority: High
Description: Users shall send real-time chat messages via WebSocket.
Inputs:

- Room ID (event ID, Long)
- User ID (from JWT in WebSocket handshake)
- Message content (string, required, max 1000 chars)

Processing:

1. Authenticate WebSocket connection via JWT interceptor
2. Extract user ID from JWT claims
3. Apply rate limiting: check Redis counter (key: "ratelimit:chat:{userId}")
4. If rate limit exceeded, reject message
5. Validate message content (not empty, length check)
6. Create ChatMessage entity:
   - Room ID: event ID
   - User ID: from JWT
   - Content: message text
   - Timestamp: now
7. Store message in Redis (key: "chat:messages:{roomId}", List data structure)
8. Broadcast message to room subscribers via STOMP: /topic/chat.{roomId}
9. Update read receipts (mark as unread for other users)
10. Increment rate limit counter

Outputs: Message broadcast to all subscribers in room
Implementation Details:

- WebSocket endpoint: /ws/chat with SockJS fallback
- STOMP destination: /app/chat.send
- Rate limiting: 10 messages per minute per user
- Message persistence: Redis List, max 1000 messages per room
- Simple broker (in-memory) for single instance, Redis pub/sub for scaling

Consequences of Failure:

- **Rate Limit Exceeded**: Message rejected, prevents spam
- **WebSocket Disconnected**: Message not delivered, user must reconnect
- **Redis Unavailable**: Messages not persisted, but real-time delivery works
- **Invalid JWT**: Connection rejected, user must re-authenticate

JML Specifications:

```
/*@ public normal_behavior
  @ requires roomId > 0;
  @ requires userId > 0;
  @ requires content != null && content.length() > 0 && content.length() <= 1000;
  @ requires !rateLimitService.isRateLimited("chat:" + userId);
  @ ensures \result != null;
  @ ensures \result.getRoomId() == roomId;
  @ ensures \result.getUserId() == userId;
  @ ensures \result.getContent().equals(content);
  @ ensures \result.getTimestamp() != null;
  @ ensures chatRepository.existsByRoomId(roomId).contains(\result);
  @ ensures rateLimitService.incrementRateLimit("chat:" + userId);
  @ also
  @ public exceptional_behavior
  @ requires content != null && (content.length() == 0 || content.length() > 1000);
  @ signals (IllegalArgumentException) true;
  @ also
  @ public exceptional_behavior
  @ requires rateLimitService.isRateLimited("chat:" + userId);
  @ signals (TooManyRequestsException) true;
  @*/
public ChatMessage sendMessage(Long roomId, Long userId, String content);
```

4.4.2 FR-CHAT-2: Retrieve Messages

Priority: Medium
Description: Users shall retrieve chat message history.

Inputs:

- Room ID (Long, required)
- Since timestamp (Instant, optional)
- Page number (integer, default: 0)
- Page size (integer, default: 50)

Processing:

1. Query messages from Redis List: "chat:messages:{roomId}"
2. Filter by timestamp if provided
3. Apply pagination
4. Return message list

Outputs: HTTP 200 OK with paginated messages
JML Specifications:

```
/*@ public normal_behavior
  @ requires roomId > 0;
  @ requires page >= 0;
  @ requires size > 0 && size <= 100;
  @ ensures \result != null;
  @ ensures \result.getContent() != null;
  @ ensures \result.getContent().size() <= size;
  @ ensures (\forall int i; 0 <= i && i < \result.getContent().size();
  @          \result.getContent().get(i).getRoomId() == roomId);
  @ ensures sinceTimestamp == null || (\forall int i; 0 <= i && i < \result.getContent().size();
  @          \result.getContent().get(i).getTimestamp().isAfter(sinceTimestamp) ||
  @          \result.getContent().get(i).getTimestamp().equals(sinceTimestamp));
  @*/
public Page<ChatMessage> retrieveMessages(Long roomId, Instant sinceTimestamp, int page, int size);
```

4.4.3 FR-CHAT-3: Track Presence

Priority: Medium
Description: System shall track online users per event room.

Inputs:

- Room ID (Long)
- User ID (from JWT)
- Action (enum: ONLINE, OFFLINE)

Processing:

1. Update Redis Set: "presence:{roomId}"
2. Add user ID if ONLINE, remove if OFFLINE
3. Broadcast presence update via STOMP: /topic/presence.{roomId}
4. Return online user count

Outputs: HTTP 200 OK with online user count
JML Specifications:

```
/*@ public normal_behavior
  @ requires roomId > 0;
  @ requires userId > 0;
  @ requires action == ONLINE || action == OFFLINE;
  @ ensures \result != null;
  @ ensures \result.getRoomId() == roomId;
  @ ensures \result.getOnlineCount() >= 0;
  @ ensures action == ONLINE ==> presenceRepository.isUserOnline(roomId, userId);
  @ ensures action == OFFLINE ==> !presenceRepository.isUserOnline(roomId, userId);
  @ ensures \result.getOnlineCount() == presenceRepository.countOnlineUsers(roomId);
  @*/
public PresenceResponse trackPresence(Long roomId, Long userId, PresenceAction action);
```

4.4.4 FR-CHAT-4: Read Receipts

Priority: Low
Description: System shall track message read receipts.
Inputs:

- Message ID (Long)
- User ID (from JWT)

Processing:

1. Store read receipt in Redis Set: "read:{messageId}"
2. Update user's read messages: "read:user:{userId}:{roomId}"

Outputs: HTTP 200 OK
JML Specifications:

```
/*@ public normal_behavior
  @ requires messageId > 0;
  @ requires userId > 0;
  @ requires chatRepository.existsById(messageId);
  @ ensures readReceiptRepository.existsByMessageIdAndUserId(messageId, userId);
  @ ensures readReceiptRepository.findByMessageId(messageId).contains(userId);
  @*/
public void markAsRead(Long messageId, Long userId);
```

4.5 Notification Service (FR-NOTIF)

4.5.1 FR-NOTIF-1: Consume Ticket Events

Priority: High
Description: System shall consume ticket events from Kafka and send notifications.
Inputs: Kafka messages from topic "ticket-events" (TICKET_CONFIRMED, TICKET_CANCELLED)
Processing:

1. Kafka consumer listens to "ticket-events" topic
2. Deserialize event payload (TicketEvent DTO)
3. Extract event type, user ID, event ID, ticket ID, amount
4. Create notification record in database
5. Send email/SMS notification (mock: log to console)
6. Trigger webhook deliveries for all active subscriptions
7. Store notification delivery status
8. Implement retry logic with exponential backoff for failed deliveries
9. Commit Kafka offset after successful processing

Implementation Details:

- Kafka consumer group: "notification-service"
- Auto-commit: false (manual commit after processing)
- Retry policy: Exponential backoff (1s, 2s, 4s, 8s, 16s, max 5 retries)
- Dead letter queue: Failed events after max retries

Consequences of Failure:

- **Kafka Consumer Lag**: Notifications delayed, but eventually processed
- **Notification Send Failure**: Retried with exponential backoff

- **Webhook Delivery Failure**: Retried up to 5 times, then marked as failed
- **Database Failure**: Kafka offset not committed, event reprocessed

JML Specifications:

```
/*@ public normal_behavior
  @ requires event != null;
  @ requires event.getEventType() == TICKET_CONFIRMED || event.getEventType() == TICKET_CANCELLED;
  @ requires event.getUserId() > 0;
  @ requires event.getEventId() > 0;
  @ ensures notificationRepository.existsByEventId(event.getEventId());
  @ ensures notificationRepository.findByEventId(event.getEventId()).getStatus() == SENT ||
  @          notificationRepository.findByEventId(event.getEventId()).getStatus() == PENDING;
  @ ensures webhookDeliveryRepository.countByEventId(event.getEventId()) >= 0;
  @*/
public void consumeTicketEvent(TicketEvent event);
```

4.5.2 FR-NOTIF-2: Register Webhook

Priority: Medium
Description: Partners shall register webhook URLs for event notifications.
Inputs:

- Partner ID (string, required)
- Webhook URL (string, required, valid URL format)

Processing:

1. Validate URL format
2. Generate unique secret (UUID)
3. Create WebhookSubscription entity
4. Set active = true
5. Persist to database
6. Return subscription ID and secret (one-time display)

Outputs: HTTP 201 Created with subscription ID and secret
Consequences of Failure:

- **Invalid URL**: Registration rejected, partner must provide valid URL
- **Duplicate Partner ID**: New subscription created, old one deactivated

JML Specifications:

```
/*@ public normal_behavior
  @ requires partnerId != null && partnerId.length() > 0;
  @ requires webhookUrl != null && webhookUrl.matches("^https?://.+");
  @ ensures \result != null;
  @ ensures \result.getId() > 0;
  @ ensures \result.getPartnerId().equals(partnerId);
  @ ensures \result.getUrl().equals(webhookUrl);
  @ ensures \result.getSecret() != null && \result.getSecret().length() > 0;
  @ ensures \result.isActive() == true;
  @ ensures webhookSubscriptionRepository.existsById(\result.getId());
  @ ensures webhookSubscriptionRepository.findByPartnerId(partnerId).isActive() == false ||
  @          webhookSubscriptionRepository.findByPartnerId(partnerId).getId() == \result.getId();
  @ also
  @ public exceptional_behavior
  @ requires webhookUrl != null && !webhookUrl.matches("^https?://.+");
  @ signals (IllegalArgumentException) true;
  @*/
public WebhookSubscriptionResponse registerWebhook(String partnerId, String webhookUrl);
```

4.5.3 FR-NOTIF-3: Deliver Webhook

Priority: High
Description: System shall deliver events to registered webhook URLs.
Inputs: Event data, subscription ID
Processing:

1. Load webhook subscription
2. Create WebhookDelivery record (status: PENDING)
3. Build HTTP POST request:
   - URL: subscription URL
   - Headers: X-Webhook-Secret: {secret}, Content-Type: application/json
   - Body: JSON event payload
4. Execute HTTP request with timeout (5 seconds)
5. If 2xx response: Update delivery status to SUCCESS
6. If non-2xx or timeout: Update status to FAILED, schedule retry
7. Store delivery attempt with timestamp and error message
8. Retry with exponential backoff: 1s, 2s, 4s, 8s, 16s (max 5 attempts)

Consequences of Failure:

- **Webhook URL Unreachable**: Retried up to 5 times, then marked as failed
- **Invalid Response**: Treated as failure, retried
- **Timeout**: Request cancelled, retried with backoff
- **Max Retries Exceeded**: Delivery marked as permanently failed, manual intervention required

JML Specifications:

```
/*@ public normal_behavior
  @ requires subscriptionId > 0;
  @ requires eventData != null;
  @ requires webhookSubscriptionRepository.existsById(subscriptionId);
  @ requires webhookSubscriptionRepository.findById(subscriptionId).isActive() == true;
  @ ensures webhookDeliveryRepository.existsBySubscriptionId(subscriptionId);
  @ ensures webhookDeliveryRepository.findBySubscriptionId(subscriptionId).getAttemptCount() >= 1;
  @ ensures webhookDeliveryRepository.findBySubscriptionId(subscriptionId).getAttemptCount() <= 5;
  @ ensures webhookDeliveryRepository.findBySubscriptionId(subscriptionId).getStatus() == SUCCESS ||
  @          webhookDeliveryRepository.findBySubscriptionId(subscriptionId).getStatus() == PENDING ||
  @          webhookDeliveryRepository.findBySubscriptionId(subscriptionId).getStatus() == FAILED;
  @ ensures webhookDeliveryRepository.findBySubscriptionId(subscriptionId).getAttemptCount() == 5 ==>
  @          webhookDeliveryRepository.findBySubscriptionId(subscriptionId).getStatus() == FAILED;
  @*/
public void deliverWebhook(Long subscriptionId, Object eventData);
```

4.5.4 FR-NOTIF-4: Get User Notifications

Priority: Medium
Description: Users shall retrieve their notification history.
Inputs: User ID (string), pagination parameters
Processing:

1. Ensure user has webhook subscription (create if doesn't exist)
2. Query delivery logs for user's subscription ID
3. Parse event payloads from JSON
4. Format as notification objects
5. Apply pagination

6. Return formatted notifications

Outputs: HTTP 200 OK with notification list
JML Specifications:

```
/*@ public normal_behavior
  @ requires userId != null && userId.length() > 0;
  @ requires page >= 0;
  @ requires size > 0 && size <= 100;
  @ ensures \result != null;
  @ ensures \result.getContent() != null;
  @ ensures \result.getContent().size() <= size;
  @ ensures (\forall int i; 0 <= i && i < \result.getContent().size();
  @          \result.getContent().get(i).getUserId().equals(userId));
  @ ensures webhookSubscriptionRepository.existsByPartnerId(userId);
  @*/
public Page<NotificationResponse> getUserNotifications(String userId, int page, int size);
```

4.6 Media Service (FR-MEDIA)

4.6.1 FR-MEDIA-1: Upload Image

Priority: Medium
Description: Users shall upload event images.
Inputs: Multipart file (image, max 10MB, formats: JPEG, PNG, GIF, WebP)
Processing:

1. Validate file type (Content-Type starts with "image/")
2. Validate file size (max 10MB)
3. Generate unique filename: UUID + original extension
4. Ensure storage directory exists
5. Save file to local storage: /app/media-storage/{filename}
6. Return CDN-like URL: http://localhost:8087/static/{filename}

Outputs: HTTP 200 OK with image URL
Consequences of Failure:

- **Invalid File Type**: Upload rejected, user must provide image
- **File Too Large**: Upload rejected, user must compress image
- **Storage Full**: Upload fails, administrator must free space

JML Specifications:

```
/*@ public normal_behavior
  @ requires file != null;
  @ requires file.getContentType() != null && file.getContentType().startsWith("image/");
  @ requires file.getSize() > 0 && file.getSize() <= 10 * 1024 * 1024; // 10MB
  @ requires file.getOriginalFilename() != null;
  @ ensures \result != null;
  @ ensures \result.getUrl() != null && \result.getUrl().length() > 0;
  @ ensures \result.getUrl().startsWith("http://localhost:8087/static/");
  @ ensures new File("/app/media-storage/" + extractFilename(\result.getUrl())).exists();
  @ also
  @ public exceptional_behavior
  @ requires file != null && (file.getContentType() == null || !file.getContentType().startsWith("image/"));
  @ signals (IllegalArgumentException) true;
  @ also
  @ public exceptional_behavior
  @ requires file != null && file.getSize() > 10 * 1024 * 1024;
  @ signals (IllegalArgumentException) true;
  @*/
public ImageUploadResponse uploadImage(MultipartFile file);
```

4.7 Analytics Service (FR-ANALYTICS)

4.7.1 FR-ANALYTICS-1: Daily Revenue

Priority: Medium
Description: System shall compute daily revenue metrics.
Inputs: Start date, end date, optional event ID
Processing:

1. Query ticket confirmations from database
2. Filter by date range and event ID if provided
3. Aggregate revenue by date: SUM(price × quantity) GROUP BY date
4. Return DailyRevenueView objects

Outputs: HTTP 200 OK with daily revenue list
JML Specifications:

```
/*@ public normal_behavior
  @ requires startDate != null;
  @ requires endDate != null;
  @ requires startDate.isBefore(endDate) || startDate.equals(endDate);
  @ ensures \result != null;
  @ ensures (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getDate().isAfter(startDate.minusDays(1)) &&
  @          \result.get(i).getDate().isBefore(endDate.plusDays(1)));
  @ ensures eventId == null || (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getEventId() == eventId);
  @ ensures (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getRevenue().compareTo(BigDecimal.ZERO) >= 0);
  @*/
public List<DailyRevenueView> dailyRevenue(LocalDate startDate, LocalDate endDate, Long eventId);
```

4.7.2 FR-ANALYTICS-2: Daily Active Users

Priority: Medium
Description: System shall compute daily active user metrics.
Inputs: Start date, end date, optional event ID
Processing:

1. Query user activity from analytics_events table
2. Filter by date range
3. Count distinct users per date
4. Return DailyActiveUsersView objects

Outputs: HTTP 200 OK with DAU list
JML Specifications:

```
/*@ public normal_behavior
  @ requires startDate != null;
  @ requires endDate != null;
  @ requires startDate.isBefore(endDate) || startDate.equals(endDate);
  @ ensures \result != null;
  @ ensures (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getDate().isAfter(startDate.minusDays(1)) &&
  @          \result.get(i).getDate().isBefore(endDate.plusDays(1)));
  @ ensures (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getActiveUsers() >= 0);
  @ ensures eventId == null || (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getEventId() == eventId);
  @*/
public List<DailyActiveUsersView> dailyActiveUsers(LocalDate startDate, LocalDate endDate, Long eventId);
```

4.7.3 FR-ANALYTICS-3: Top Events

Priority: Medium
Description: System shall identify top performing events.
Inputs: Number of days (integer, default: 7), limit (integer, default: 5)
Processing:

1. Query ticket sales for date range
2. Aggregate by event ID: COUNT(tickets), SUM(revenue)
3. Sort by sales count or revenue (descending)
4. Return top N events

Outputs: HTTP 200 OK with top events list
JML Specifications:

```
/*@ public normal_behavior
  @ requires numberOfDays > 0;
  @ requires limit > 0 && limit <= 100;
  @ ensures \result != null;
  @ ensures \result.size() <= limit;
  @ ensures (\forall int i; 0 <= i && i < \result.size() - 1;
  @          \result.get(i).getSalesCount() >= \result.get(i+1).getSalesCount() ||
  @          \result.get(i).getRevenue().compareTo(\result.get(i+1).getRevenue()) >= 0);
  @ ensures (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getEventId() > 0);
  @ ensures (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getSalesCount() >= 0);
  @ ensures (\forall int i; 0 <= i && i < \result.size();
  @          \result.get(i).getRevenue().compareTo(BigDecimal.ZERO) >= 0);
  @*/
public List<TopEventView> topEvents(int numberOfDays, int limit);
```

4.7.4 FR-ANALYTICS-4: Dashboard Summary

Priority: Medium
Description: System shall provide combined analytics dashboard data.
Inputs: Number of days (integer, default: 7), top events count (integer, default: 5)
Processing:

1. Compute daily revenue
2. Compute daily active users
3. Compute top events
4. Combine into summary object

Outputs: HTTP 200 OK with dashboard summary
JML Specifications:

```
/*@ public normal_behavior
  @ requires numberOfDays > 0;
  @ requires topEventsCount > 0 && topEventsCount <= 100;
  @ ensures \result != null;
  @ ensures \result.getDailyRevenue() != null;
  @ ensures \result.getDailyActiveUsers() != null;
  @ ensures \result.getTopEvents() != null;
  @ ensures \result.getTopEvents().size() <= topEventsCount;
  @ ensures \result.getDailyRevenue().size() == numberOfDays;
  @ ensures \result.getDailyActiveUsers().size() == numberOfDays;
  @ ensures (\forall int i; 0 <= i && i < \result.getDailyRevenue().size();
  @          \result.getDailyRevenue().get(i).getDate() != null);
  @ ensures (\forall int i; 0 <= i && i < \result.getDailyActiveUsers().size();
  @          \result.getDailyActiveUsers().get(i).getDate() != null);
  @*/
public DashboardSummary dashboardSummary(int numberOfDays, int topEventsCount);
```

5 Data Models and Relationships

5.1 User Entity

- **Table**: users
- **Fields**: id (BIGSERIAL PK), email (VARCHAR(255) UNIQUE), password_hash (VARCHAR(255)), roles (VARCHAR[]), created_at (TIMESTAMPTZ), updated_at (TIMESTAMPTZ)
- **Relationships**: One-to-many with Tickets, Events (as organizer)

5.2 Event Entity

- **Table**: events
- **Fields**: id (BIGSERIAL PK), title (VARCHAR(140)), description (TEXT), city (VARCHAR(64)), event_time (TIMESTAMPTZ), capacity (INTEGER), organizer_id (BIGINT FK), venue (VARCHAR(255)), category (VARCHAR(64)), image_url (VARCHAR(512)), is_public (BOOLEAN), price (NUMERIC(12,2)), version (BIGINT), popularity_score (BIGINT)
- **Indexes**: idx_events_city_time (city, event_time), idx_events_organizer_id (organizer_id)
- **Relationships**: Many-to-one with User (organizer), One-to-many with Tickets, One-to-one with SeatInventory

5.3 Ticket Entity

- **Table**: tickets
- **Fields**: id (UUID PK), event_id (BIGINT FK), user_id (BIGINT FK), status (VARCHAR), price (NUMERIC(12,2)), quantity (INTEGER), idempotency_key (VARCHAR UNIQUE), locked_at (TIMESTAMPTZ), lock_expires_at (TIMESTAMPTZ), created_at (TIMESTAMPTZ), updated_at (TIMESTAMPTZ), version (BIGINT)
- **Indexes**: idx_tickets_event (event_id), idx_tickets_user (user_id), idx_tickets_idempotency (idempotency_key)
- **Relationships**: Many-to-one with Event, Many-to-one with User

5.4 SeatInventory Entity

- **Table**: seat_inventory
- **Fields**: event_id (BIGINT PK/FK), total_seats (INTEGER), available_seats (INTEGER), version (BIGINT)
- **Relationships**: One-to-one with Event

5.5 WebhookSubscription Entity

- **Table**: webhook_subscriptions
- **Fields**: id (BIGSERIAL PK), partner_id (VARCHAR), url (VARCHAR), secret (VARCHAR), active (BOOLEAN), created_at (TIMESTAMPTZ), updated_at (TIMESTAMPTZ)
- **Relationships**: One-to-many with WebhookDelivery

5.6 WebhookDelivery Entity

- **Table**: webhook_deliveries
- **Fields**: id (BIGSERIAL PK), subscription_id (BIGINT FK), domain_event_type (VARCHAR), domain_event_id (VARCHAR), payload (JSONB), status (VARCHAR), attempt_count (INTEGER), last_error (TEXT), next_retry_at (TIMESTAMPTZ), created_at (TIMESTAMPTZ)
- **Relationships**: Many-to-one with WebhookSubscription

6 Interface Requirements

6.1 User Interfaces

- **Web Application**: Responsive Next.js application supporting desktop (1024x768+) and mobile (320x568+) viewports
- **Admin Dashboard**: Manager interface for event organizers and administrators
- **API Documentation**: REST API endpoints (future: Swagger/OpenAPI UI)

6.2 Hardware Interfaces

- Standard x86_64 server hardware for containerized services
- Network interfaces for inter-service communication (Docker bridge network)
- Storage volumes for media files and database persistence

6.3 Software Interfaces

- **PostgreSQL 16**: JDBC connection via HikariCP connection pool
- **Redis 7**: Jedis/Lettuce client via Spring Data Redis
- **Kafka 7.6.0**: Spring Kafka for producer/consumer
- **Elasticsearch**: Elasticsearch Java Client
- **Nginx**: HTTP reverse proxy configuration
- **Spring Cloud Gateway**: Route configuration via application.properties

6.4 Communication Interfaces

- **HTTP/HTTPS**: REST API communication (port 80/443 via Nginx, 8085 via Gateway)
- **WebSocket**: Real-time chat communication (port 8086, path: /ws/chat)
- **STOMP**: WebSocket sub-protocol for message routing
- **TCP/IP**: Inter-service communication within Docker network
- **Kafka Protocol**: Message queue communication (port 9092)

7 Performance Requirements

7.1 Response Time Requirements

- **API Endpoints**: 95% of requests shall respond within 500ms, 99% within 1s
- **Event List Queries**: Paginated lists shall load within 300ms (cached) or 800ms (uncached)
- **Event Details**: Cached responses within 50ms, uncached within 200ms
- **Search Queries**: Elasticsearch queries shall complete within 400ms
- **WebSocket Messages**: Real-time message delivery within 100ms
- **Webhook Delivery**: Initial attempt within 1 second, retries with exponential backoff
- **Ticket Locking**: Complete within 200ms under normal load
- **Ticket Confirmation**: Complete within 300ms including Kafka publish

7.2 Throughput Requirements

- **Concurrent Users**: Support 1,000 concurrent authenticated users
- **Requests per Second**: Handle 500 requests/second per service instance
- **Ticket Bookings**: Process 100 ticket confirmations per second
- **Chat Messages**: Handle 200 messages per second per room
- **Webhook Deliveries**: Process 50 webhook deliveries per second
- **Event Searches**: Handle 200 search queries per second

7.3 Resource Utilization

- **CPU**: Average CPU usage per service instance shall not exceed 70%
- **Memory**: Each service instance shall use maximum 512MB RAM
- **Disk**: Database growth rate: 1GB per 100,000 events
- **Network**: Bandwidth usage: 10Mbps per 100 concurrent users
- **Database Connections**: Maximum 20 connections per service instance

7.4 Scalability Requirements

- System shall support horizontal scaling of individual microservices
- Database shall support up to 10 million events
- Cache shall handle 1 million concurrent keys
- Message queue shall process 10,000 messages per second
- Elasticsearch shall index 1 million events with sub-second search response

8 Design Constraints

8.1 Standards Compliance

- RESTful API design following OpenAPI 3.0 specification
- JWT token format per RFC 7519
- HTTP/1.1 and HTTP/2 protocols
- WebSocket protocol per RFC 6455
- SQL:2016 standard for database queries
- Docker containerization standards (OCI image format)
- JSON data format per RFC 7159

8.2 Hardware Limitations

- Minimum 4GB RAM per service instance
- Minimum 2 CPU cores per service instance
- Network latency: Maximum 100ms between services
- Storage: Minimum 100GB for media files
- Disk I/O: Minimum 100 IOPS for database

8.3 Software Constraints

- Java 17+ runtime environment
- Spring Boot 3.x framework
- Node.js 18+ for frontend applications
- PostgreSQL 16+ database
- Redis 7+ cache
- Docker 20.10+ container runtime
- Linux kernel 5.4+ for container support

8.4 Other Constraints

- All services must be stateless (except chat service WebSocket connections)
- Database transactions must maintain ACID properties
- No direct database access from frontend applications
- All external communication must go through API Gateway
- Media files must be served via CDN simulation (Nginx)
- JWT tokens must expire within 24 hours
- Rate limiting must be applied to prevent abuse

9 Non-Functional Attributes

9.1 Reliability

- **System Uptime**: 99.5% availability (43.8 hours downtime per year)
- **Mean Time Between Failures (MTBF)**: Minimum 720 hours (30 days)
- **Mean Time To Recovery (MTTR)**: Maximum 15 minutes
- **Error Rate**: Less than 0.1% of requests result in 5xx errors
- **Data Integrity**: Zero data loss for confirmed ticket bookings
- **Fault Tolerance**: System shall continue operating if one non-critical service fails
- **Graceful Degradation**: Cache failures shall not prevent core functionality

9.2 Security

- **Authentication**: JWT-based authentication with 24-hour token expiration
- **Authorization**: Role-based access control (USER, ORGANIZER, ADMIN)
- **Password Security**: BCrypt hashing with salt rounds (10)
- **HTTPS**: All external communication encrypted via TLS 1.2+
- **Input Validation**: All user inputs validated and sanitized
- **SQL Injection Prevention**: Parameterized queries via JPA
- **XSS Prevention**: Content Security Policy headers
- **CSRF Protection**: Token-based CSRF protection
- **Rate Limiting**: Per-user and per-IP rate limiting
- **Secrets Management**: JWT secrets stored in environment variables
- **Webhook Security**: HMAC-based authentication for webhook deliveries
- **Data Encryption**: Sensitive data encrypted at rest (future requirement)

9.3 Maintainability

- **Code Organization**: Microservices architecture with clear separation of concerns
- **Documentation**: Inline code comments and API documentation
- **Logging**: Structured logging with correlation IDs (future)
- **Monitoring**: Spring Actuator health checks on all services
- **Version Control**: Git-based version control with semantic versioning
- **Testing**: Unit tests, integration tests, and end-to-end tests
- **Code Quality**: Code reviews, static analysis tools

9.4 Portability

- **Containerization**: All services containerized via Docker
- **Platform Independence**: Java-based services run on any OS supporting Java 17+
- **Database Portability**: PostgreSQL can be replaced with other SQL databases
- **Cloud Compatibility**: Designed for deployment on AWS, Azure, GCP
- **Local Development**: docker-compose enables local development environment

9.5 Usability

- **User Interface**: Intuitive, responsive design following Material Design principles
- **Error Messages**: Clear, user-friendly error messages
- **Loading States**: Visual feedback during async operations
- **Accessibility**: WCAG 2.1 Level AA compliance (future)
- **Multi-language**: Support for English (future: i18n)
- **Mobile Responsive**: Optimized for mobile devices
- **Help Documentation**: User guides and API documentation

9.6 Scalability

- **Horizontal Scaling**: Each microservice can scale independently
- **Database Scaling**: Read replicas for read-heavy operations
- **Cache Scaling**: Redis cluster support (future)
- **Load Balancing**: Nginx and API Gateway support load balancing
- **Sharding**: Event service designed for database sharding by city
- **Message Queue Scaling**: Kafka supports partition scaling

9.7 Performance

- **Caching Strategy**: Multi-layer caching (Redis, application-level, CDN)
- **Database Optimization**: Indexed queries, connection pooling
- **Async Processing**: Kafka-based async event processing
- **Bloom Filter**: Fast existence checks before database queries
- **Pagination**: All list endpoints support pagination
- **Lazy Loading**: Frontend implements lazy loading for images

9.8 Availability

- **Health Checks**: Spring Actuator health endpoints on all services
- **Circuit Breakers**: Resilience patterns for service failures (future)
- **Retry Logic**: Exponential backoff for webhook deliveries
- **Idempotency**: Idempotent operations prevent duplicate processing
- **Database Replication**: PostgreSQL replication for high availability (future)
- **Backup Strategy**: Daily database backups (future)

9.9 Interoperability

- **API Standards**: RESTful APIs following OpenAPI specification
- **Data Formats**: JSON for all API communication
- **Web Standards**: HTTP/HTTPS, WebSocket standards compliance
- **Database Standards**: SQL standard compliance
- **Integration**: Webhook system enables third-party integrations

9.10 Testability

- **Unit Testing**: JUnit tests for business logic
- **Integration Testing**: Spring Boot Test for service integration
- **API Testing**: Postman/curl scripts for API validation
- **Test Data**: Isolated test databases per service
- **Mocking**: Mock external dependencies in tests

9.11 Reusability

- **Shared Libraries**: Common DTOs and utilities
- **Service Patterns**: Consistent service layer patterns
- **Component Library**: Reusable React components
- **API Contracts**: Well-defined API contracts for service communication

10 Error Handling and Exception Management

10.1 Exception Hierarchy

The system implements a hierarchical exception handling strategy:

- **GlobalExceptionHandler**: @RestControllerAdvice catches all exceptions
- **EntityNotFoundException**: 404 Not Found for missing resources
- **IllegalArgumentException**: 400 Bad Request for invalid input
- **IllegalStateException**: 400 Bad Request for invalid state transitions
- **OptimisticLockException**: 409 Conflict for concurrent update conflicts
- **MethodArgumentNotValidException**: 400 Bad Request for validation failures
- **ResponseStatusException**: Custom HTTP status codes
- **Generic Exception**: 500 Internal Server Error for unexpected errors

10.2 Error Response Format

All error responses follow consistent format:

```
{
  "timestamp": "2024-01-15T10:30:00Z",
  "status": 400,
  "error": "Bad Request",
  "message": "Field-specific error message",
  "path": "/api/events"
}
```

10.3 Error Handling by Service

Auth Service:

- Validation errors: Field-specific messages
- Duplicate email: 409 Conflict
- Invalid credentials: 401 Unauthorized (generic message to prevent enumeration)
- Rate limit exceeded: 429 Too Many Requests

Event Service:

- Entity not found: 404 Not Found
- Validation failure: 400 Bad Request with field errors
- Unauthorized access: 403 Forbidden
- Optimistic lock conflict: 409 Conflict
- Elasticsearch failure: Logged but doesn't fail request

Ticket Service:

- Insufficient seats: 400 Bad Request
- Lock expired: 400 Bad Request
- Invalid ticket state: 400 Bad Request
- Idempotency conflict: 409 Conflict (but returns existing ticket)
- Optimistic lock conflict: 409 Conflict

Chat Service:

- Rate limit exceeded: Message rejected, error sent via WebSocket
- Invalid JWT: Connection rejected
- Message validation failure: Error response via STOMP

Notification Service:

- Webhook delivery failure: Retried with exponential backoff
- Kafka consumer failure: Offset not committed, event reprocessed
- Database failure: Transaction rolled back, event remains in Kafka

10.4 Error Recovery Strategies

- **Transient Failures**: Automatic retry with exponential backoff
- **Permanent Failures**: Logged for manual intervention
- **Partial Failures**: Transaction rollback ensures consistency
- **Cache Failures**: Graceful degradation to database queries
- **Service Failures**: Health checks trigger alerts

11 Security Requirements

11.1 Authentication Requirements

- All API endpoints (except /auth/register, /auth/login) require JWT authentication
- JWT tokens must be validated on every request
- Token expiration: 24 hours from issue time
- Token refresh: Not implemented (user must re-login)
- Password requirements: Minimum 8 characters
- Password hashing: BCrypt with salt rounds = 10

11.2 Authorization Requirements

- Role-based access control: USER, ORGANIZER, ADMIN
- Event creation: Requires ORGANIZER or ADMIN role
- Event update/delete: Only event organizer or ADMIN
- Admin endpoints: Require ADMIN role or bypass token
- Webhook registration: No authentication required (future: API key)

11.3 Data Protection Requirements

- Passwords: Never stored in plain text, hashed with BCrypt
- JWT secrets: Stored in environment variables, not in code
- Webhook secrets: Generated UUIDs, displayed once
- Sensitive data: Not logged in application logs
- Data encryption: At rest (future requirement), in transit (TLS)

11.4 Input Validation Requirements

- All user inputs validated using Bean Validation annotations
- SQL injection prevention: Parameterized queries via JPA
- XSS prevention: Input sanitization (future: output encoding)
- File upload validation: Type and size checks
- URL validation: Webhook URLs must be valid HTTP/HTTPS URLs

11.5 Rate Limiting Requirements

- Login endpoint: 5 requests per minute per IP
- Chat messages: 10 messages per minute per user
- General API: 20 requests per second per user (if enabled)
- Rate limit storage: Redis with TTL
- Rate limit exceeded: HTTP 429 Too Many Requests

12 Deployment and Operations

12.1 Deployment Architecture

- **Containerization**: All services containerized via Docker
- **Orchestration**: docker-compose for local development
- **Service Discovery**: Docker DNS-based service discovery
- **Load Balancing**: Nginx reverse proxy, API Gateway routing
- **Scaling**: Horizontal scaling via multiple container instances

12.2 Configuration Management

- **Environment Variables**: Database URLs, secrets, service ports
- **Application Properties**: Service-specific configuration
- **Docker Profiles**: Separate profiles for local and docker environments
- **Secrets**: JWT secrets, database passwords via environment variables

12.3 Monitoring and Observability

- **Health Checks**: Spring Actuator /actuator/health endpoints
- **Metrics**: Spring Actuator metrics (future: Prometheus)
- **Logging**: Application logs to stdout (future: structured logging)
- **Tracing**: Not implemented (future: distributed tracing)
- **Alerting**: Not implemented (future: alert manager)

12.4 Backup and Recovery

- **Database Backups**: Not automated (future: daily backups)
- **Media Storage**: Local filesystem (future: object storage)
- **Disaster Recovery**: Not implemented (future requirement)
- **Data Retention**: Indefinite (future: retention policies)

13 Appendices

13.1 Acronyms and Abbreviations

Acronym	Definition
SRS	Software Requirements Specification
API	Application Programming Interface
JWT	JSON Web Token
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
WebSocket	Web Socket Protocol
STOMP	Simple Text Oriented Messaging Protocol
SQL	Structured Query Language
ACID	Atomicity, Consistency, Isolation, Durability
CAP	Consistency, Availability, Partition tolerance
TTL	Time To Live
LRU	Least Recently Used
CDN	Content Delivery Network
DAU	Daily Active Users
MAU	Monthly Active Users
MTBF	Mean Time Between Failures
MTTR	Mean Time To Recovery
XSS	Cross-Site Scripting
CSRF	Cross-Site Request Forgery
HMAC	Hash-based Message Authentication Code
WCAG	Web Content Accessibility Guidelines
i18n	Internationalization
JSON	JavaScript Object Notation
UUID	Universally Unique Identifier

13.2 Glossary

- **Event**: A scheduled occurrence that users can attend and purchase tickets for
- **Organizer**: User who creates and manages events
- **Ticket**: A reservation for attending an event
- **Lock**: Temporary reservation of seats before confirmation (10-minute duration)
- **Idempotency**: Property of operations that can be applied multiple times without changing the result
- **Webhook**: HTTP callback for delivering events to external systems
- **Microservice**: Independent, deployable service component
- **Sharding**: Horizontal partitioning of data across multiple databases
- **Bloom Filter**: Probabilistic data structure for fast existence checks
- **Consistent Hashing**: Hashing technique for distributed caching
- **Presence**: Real-time tracking of online users in a chat room
- **Feed**: Personalized list of recommended events
- **Optimistic Locking**: Concurrency control using version fields
- **Cache-Aside**: Caching pattern where application manages cache
- **Dead Letter Queue**: Queue for messages that failed processing after max retries

13.3 References

- IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications
- Spring Boot Documentation: `https://spring.io/projects/spring-boot`
- Next.js Documentation: `https://nextjs.org/docs`
- PostgreSQL Documentation: `https://www.postgresql.org/docs/`
- Redis Documentation: `https://redis.io/documentation`
- Apache Kafka Documentation: `https://kafka.apache.org/documentation/`
- Docker Documentation: `https://docs.docker.com/`
- JWT Specification: RFC 7519
- WebSocket Specification: RFC 6455
- REST Architectural Style: Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures
- OpenAPI Specification: `https://swagger.io/specification/`

13.4 Requirement Traceability Matrix

Requirement ID	Service	Implementation Location
FR-AUTH-1	auth-service	AuthController.register(), AuthService.register()
FR-AUTH-2	auth-service	AuthController.login(), AuthService.login()
FR-AUTH-3	auth-service	AuthController.getCurrentUser(), AuthService.getCurrentUser()
FR-EVENT-1	event-service	EventController.createEvent(), EventService.createEvent()
FR-EVENT-2	event-service	EventController.listEvents(), EventService.searchEvents()
FR-EVENT-3	event-service	EventController.getEvent(), EventService.getEventById()
FR-EVENT-4	event-service	EventController.search(), EventSearchService.search()
FR-EVENT-5	event-service	EventController.updateEvent(), EventService.updateEvent()
FR-EVENT-6	event-service	EventController.deleteEvent(), EventService.deleteEvent()
FR-EVENT-7	event-service	FeedController.getFeed(), FeedService.getFeed()
FR-TICKET-1	ticket-service	TicketController.lock(), TicketService.lockTickets()
FR-TICKET-2	ticket-service	TicketController.confirm(), TicketService.confirmTicket()
FR-TICKET-3	ticket-service	TicketController.cancel(), TicketService.cancelTicket()
FR-TICKET-4	ticket-service	TicketController.myTickets(), TicketService.listTicketsForUser()
FR-TICKET-5	ticket-service	TicketController.availability(), TicketService.getSeatAvailability()
FR-CHAT-1	chat-service	ChatSocketController.sendMessage(), ChatRepository.save()
FR-CHAT-2	chat-service	ChatController.getMessages(), ChatRepository.findByRoomId()
FR-CHAT-3	chat-service	PresenceController.updatePresence(), PresenceService.trackPresence()

FR-CHAT-4	chat-service	ReadReceiptController.markRead(), ReadReceiptRepository.save()
FR-NOTIF-1	notification-service	TicketEventListener.onTicketEvent(), NotificationService.sendNotification()
FR-NOTIF-2	notification-service	WebhookController.register(), WebhookSubscriptionRepository.save()
FR-NOTIF-3	notification-service	DeliveryService.deliverWebhook(), WebhookDeliveryRepository.save()
FR-NOTIF-4	notification-service	WebhookController.getUserNotifications(), UserWebhookService.getNotifications()
FR-MEDIA-1	media-service	MediaController.upload(), MediaService.saveFile()
FR-ANALYTICS-1	analytics-service	AnalyticsDashboardController.dailyRevenue(), AnalyticsQueryService.dailyRevenue()
FR-ANALYTICS-2	analytics-service	AnalyticsDashboardController.dailyActiveUsers(), AnalyticsQueryService.dailyActiveUsers()
FR-ANALYTICS-3	analytics-service	AnalyticsDashboardController.topEvents(), AnalyticsQueryService.topEvents()
FR-ANALYTICS-4	analytics-service	AnalyticsDashboardController.dashboard(), AnalyticsQueryService.dashboardSummary()