Anti-Scalping Ticket Platform - MVP Monolith Guide

🎯 Project Mission

Eliminate ticket scalping and black market reselling through a secure, platform-locked ticket exchange system with price controls and fair redistribution.

🪜 MVP Tech Stack (Monolith - Latest Versions)

Core Technologies

- **Java 25** (Latest - March 2025 release with virtual threads, pattern matching enhancements)
- **Spring Framework 7.x** (Latest major version)
- **Spring Boot 4.x** (Built on Spring 7)
- **Build Tool:** Maven
- **Database:** PostgreSQL 17+ (Latest stable)
- **Cache:** Redis 7.x (Latest stable)
- **Security:** Spring Security 7 + JWT
- **API:** RESTful APIs
- **Message Queue:** RabbitMQ 4.x (optional for MVP, good for pool notifications)

Why This Cutting-Edge Stack?

Java 25 Advantages:

- **Virtual Threads (Project Loom):** Handle thousands of concurrent pool claims efficiently without traditional thread overhead
- **Pattern Matching:** Cleaner, more readable transfer validation logic
- **Record Patterns:** Perfect for immutable DTOs and request/response objects
- **String Templates:** Easier dynamic query building
- **Performance:** Latest JVM optimizations for high-throughput ticketing scenarios

Spring Boot 4 + Spring 7 Benefits:

- **Native Compilation:** Can compile to native binary with GraalVM for faster startup
- **Enhanced Observability:** Built-in metrics and tracing for monitoring pool activity and transfer patterns
- **HTTP Interface Clients:** Cleaner external API calls (payment gateways, venue APIs)
- **Problem Details (RFC 7807):** Standardized error responses
- **Virtual Threads Integration:** Seamless support for Java 25's virtual threads

Maven Benefits:

- Industry standard, battle-tested
- Excellent plugin ecosystem
- Dependency management
- Strong IDE integration (IntelliJ, Eclipse, VS Code)

Development Tools

- **IDE:** IntelliJ IDEA / VS Code with Java extensions
- **Version Control:** Git + GitHub/GitLab
- **Testing:** JUnit 5, Mockito, TestContainers
- **API Documentation:** SpringDoc OpenAPI 3 (Swagger)
- **Monitoring:** Spring Boot Actuator + Micrometer
- **Containerization:** Docker + Docker Compose (for PostgreSQL, Redis)

⚠️ Considerations with Latest Stack

Pros:

- ✅ Future-proof - won't need major upgrades for years
- ✅ Best performance and latest features
- ✅ Showcases cutting-edge technical skills
- ✅ Virtual threads perfect for concurrent operations
- ✅ Modern, clean code patterns
- ✅ Great for portfolio/open-source credibility

Potential Challenges:

- ⚠️ Some third-party libraries might need compatibility testing

- ⚠️ Less Stack Overflow content for newest features
- ⚠️ Early adopter risk (though Spring Boot 4 is stable)
- ⚠️ Contributors might need time to learn latest features

📦 MVP Scope - Phase 1 (Unseated Events Only)

Core Features to Build

1. User Management

- User registration with email/phone verification
- Login/logout with JWT tokens (Spring Security 7)
- Profile management
- Identity verification (basic: email/phone, advanced: ID upload)
- Trusted Circle management (add/remove 3-5 trusted contacts)
- Account status (active, suspended, banned)

2. Event Management

- Admin can create events (name, date, venue, capacity, price)
- Event listing page with filters
- Event details page
- Unseated/general admission only for MVP
- Event status management (upcoming, live, completed, cancelled)

3. Ticket Purchase

- Browse available events
- Purchase tickets at retail price
- Purchase limits per user/event (e.g., max 4 tickets)
- Payment integration (Stripe or Razorpay)
- Digital ticket generation with unique QR code
- Purchase confirmation and receipt

4. Ticket Ownership & Display

- "My Tickets" dashboard
- Dynamic QR code display (refreshes every 30-60 seconds using virtual threads)

- Ticket status: active, transferred, cancelled, used
- Basic screenshot detection warning
- Ticket details (event info, purchase date, transfer history)

5. Hybrid Transfer System

A. Trusted Circle Transfer

- Instant transfer to trusted circle members (≤5 people)
- No fees, no time restrictions
- Both parties must be verified
- Transfer history tracked
- Unlimited transfers within circle

B. One-Time Controlled Transfer

- Transfer to non-circle user (max 1-2 times per ticket)
- Must happen 48+ hours before event
- Both parties verify identity
- Select transfer reason from dropdown (gift, can't attend, schedule conflict, etc.)
- Email confirmation to both parties
- Transfer history and pattern tracking for fraud detection

C. After Transfer Limit Reached

- Only option: cancel ticket to pool
- Can nominate someone for priority claim

6. Time-Based Cancellation System

- User cancels ticket through dashboard
- Refund calculated dynamically based on time remaining:
  - **24+ hours before event:** 90% refund
  - **12-24 hours:** 70% refund
  - **6-12 hours:** 50% refund
  - **< 6 hours:** 25% refund
- Retained amount goes to platform/organizer (configurable split)
- Ticket automatically goes to pool
- Cancellation confirmation email

- Refund processed within 5-7 business days

7. Ticket Pool Marketplace

- Dashboard showing all available pool tickets
- Real-time updates when tickets added (WebSocket or Server-Sent Events)
- Filter by event, date, price
- First-come-first-served claiming (lottery optional for v2)
- Claim ticket at original retail price only
- **15-minute nomination priority system:**
  - Person cancelling can nominate one user
  - Nominee gets 15-minute exclusive claim window
  - Email/SMS notification to nominee
  - After expiry, ticket goes to public pool
- Claimed ticket transferred to new owner instantly
- Transaction history and audit trail

8. Price Cap Enforcement

- All pool tickets priced at original retail value
- System prevents any markup in code and database constraints
- Transparent pricing display
- Price history visible to users

9. Anti-Scalping Controls

- Purchase limits per user (configurable per event)
- Purchase limits per payment method
- Pattern detection for suspicious transfers
- Velocity checks (too many purchases in short time)
- Account flagging system with severity levels
- Ban/suspend accounts that violate rules
- Admin dashboard for reviewing flagged accounts
- IP-based rate limiting

10. Venue Entry System

- QR code scanner simulation (for MVP testing)

- Real venue scanner API integration ready
- Verify ticket validity in real-time
- Check ticket status, ownership, event match
- Mark ticket as "used" (one-time entry)
- Handle edge cases (duplicate scans, invalid tickets)
- Entry logs for audit trail

🗂️ Monolith Package Structure

```
com.antiscalping.tickets
|
├── config/                          # Spring configurations
|   ├── SecurityConfig.java          # Spring Security 7 + JWT
|   ├── RedisConfig.java             # Redis connection and caching
|   ├── OpenApiConfig.java           # Swagger/OpenAPI documentation
|   ├── AsyncConfig.java             # Virtual threads executor config
|   └── WebSocketConfig.java         # Real-time pool updates
|
├── model/                           # Domain entities (use Records where
appropriate)
|   ├── User.java
|   ├── Event.java
|   ├── Ticket.java
|   ├── Transfer.java
|   ├── Cancellation.java
|   ├── TrustedCircle.java
|   ├── PoolTicket.java
|   ├── Payment.java
|   └── AuditLog.java
|
├── repository/                      # JPA repositories
|   ├── UserRepository.java
|   ├── EventRepository.java
|   ├── TicketRepository.java
|   ├── TransferRepository.java
|   ├── PoolTicketRepository.java
|   ├── PaymentRepository.java
|   └── AuditLogRepository.java
|
├── service/                         # Business logic
|   ├── UserService.java
|   ├── AuthService.java
|   ├── EventService.java
```

```
|   ├── TicketService.java
|   ├── TransferService.java
|   ├── CancellationService.java
|   ├── PoolService.java            # Uses virtual threads for concurrent claims
|   ├── PricingService.java
|   ├── QRCodeService.java          # Dynamic QR generation with Redis
|   ├── FraudDetectionService.java
|   ├── PaymentService.java
|   ├── NotificationService.java    # Email/SMS notifications
|   └── AnalyticsService.java
|
├── controller/                     # REST endpoints
|   ├── AuthController.java
|   ├── UserController.java
|   ├── EventController.java
|   ├── TicketController.java
|   ├── TransferController.java
|   ├── PoolController.java
|   ├── AdminController.java
|   └── ScannerController.java
|
├── dto/                            # Data transfer objects (use Java Records)
|   ├── request/
|   |   ├── LoginRequest.java
|   |   ├── RegisterRequest.java
|   |   ├── TicketPurchaseRequest.java
|   |   ├── TransferRequest.java
|   |   └── CancellationRequest.java
|   └── response/
|       ├── AuthResponse.java
|       ├── TicketResponse.java
|       ├── EventResponse.java
|       └── PoolTicketResponse.java
|
├── exception/                      # Custom exceptions
|   ├── TicketNotFoundException.java
|   ├── TransferNotAllowedException.java
|   ├── InsufficientRefundException.java
|   ├── PoolClaimException.java
|   └── GlobalExceptionHandler.java # Centralized exception handling
|
├── security/                       # Security components
|   ├── JwtTokenProvider.java
|   ├── JwtAuthenticationFilter.java
|   ├── UserDetailsServiceImpl.java
|   └── SecurityConstants.java
|
```

```
├── util/                          # Utility classes
│   ├── QRCodeGenerator.java       # Uses Google ZXing
│   ├── DateTimeUtil.java
│   ├── ValidationUtil.java
│   └── CryptoUtil.java            # For QR signature generation
│
└── TicketPlatformApplication.java  # Main Spring Boot application
```

Core Database Schema (PostgreSQL 17)

Users Table

sql

```sql
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) UNIQUE NOT NULL,
    phone VARCHAR(20) UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    verified BOOLEAN DEFAULT FALSE,
    account_status VARCHAR(20) DEFAULT 'ACTIVE', -- ACTIVE, SUSPENDED, BANNED
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_users_phone ON users(phone);
```

Events Table

sql

```sql
CREATE TABLE events (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    description TEXT,
    venue VARCHAR(255) NOT NULL,
    event_date TIMESTAMP NOT NULL,
    capacity INTEGER NOT NULL,
    tickets_sold INTEGER DEFAULT 0,
    ticket_price DECIMAL(10, 2) NOT NULL,
    event_type VARCHAR(20) DEFAULT 'UNSEATED', -- UNSEATED, SEATED
    status VARCHAR(20) DEFAULT 'UPCOMING', -- UPCOMING, LIVE, COMPLETED, CANCELLED
    created_by UUID REFERENCES users(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT check_capacity CHECK (tickets_sold <= capacity),
    CONSTRAINT check_price CHECK (ticket_price > 0)
);

CREATE INDEX idx_events_date ON events(event_date);
CREATE INDEX idx_events_status ON events(status);
```

Tickets Table

```sql
```

```sql
CREATE TABLE tickets (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    event_id UUID NOT NULL REFERENCES events(id) ON DELETE CASCADE,
    owner_id UUID NOT NULL REFERENCES users(id),
    original_buyer_id UUID NOT NULL REFERENCES users(id),
    purchase_price DECIMAL(10, 2) NOT NULL,
    status VARCHAR(20) DEFAULT 'ACTIVE', -- ACTIVE, TRANSFERRED, CANCELLED, USED
    qr_code_seed VARCHAR(255) NOT NULL UNIQUE, -- For dynamic QR generation
    transfer_count INTEGER DEFAULT 0,
    max_transfers INTEGER DEFAULT 2,
    purchase_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT check_transfer_count CHECK (transfer_count <= max_transfers)
);

CREATE INDEX idx_tickets_owner ON tickets(owner_id);
CREATE INDEX idx_tickets_event ON tickets(event_id);
CREATE INDEX idx_tickets_status ON tickets(status);
```

Transfers Table

sql

```sql
CREATE TABLE transfers (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    ticket_id UUID NOT NULL REFERENCES tickets(id) ON DELETE CASCADE,
    from_user_id UUID NOT NULL REFERENCES users(id),
    to_user_id UUID NOT NULL REFERENCES users(id),
    transfer_type VARCHAR(30) NOT NULL, -- TRUSTED_CIRCLE, CONTROLLED
    transfer_reason VARCHAR(100),
    transfer_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    verified BOOLEAN DEFAULT FALSE,
    hours_before_event INTEGER,

    CONSTRAINT check_different_users CHECK (from_user_id != to_user_id)
);

CREATE INDEX idx_transfers_ticket ON transfers(ticket_id);
CREATE INDEX idx_transfers_from ON transfers(from_user_id);
CREATE INDEX idx_transfers_to ON transfers(to_user_id);
```

Trusted_Circles Table

```sql
CREATE TABLE trusted_circles (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    trusted_user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    added_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20) DEFAULT 'ACTIVE', -- ACTIVE, REMOVED

    CONSTRAINT unique_circle_pair UNIQUE (user_id, trusted_user_id),
    CONSTRAINT check_not_self CHECK (user_id != trusted_user_id)
);

CREATE INDEX idx_trusted_circles_user ON trusted_circles(user_id);
```

Cancellations Table

```sql
CREATE TABLE cancellations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    ticket_id UUID NOT NULL REFERENCES tickets(id) ON DELETE CASCADE,
    user_id UUID NOT NULL REFERENCES users(id),
    event_id UUID NOT NULL REFERENCES events(id),
    cancellation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    event_date TIMESTAMP NOT NULL,
    hours_before_event INTEGER NOT NULL,
    refund_percentage INTEGER NOT NULL,
    refund_amount DECIMAL(10, 2) NOT NULL,
    retained_amount DECIMAL(10, 2) NOT NULL,
    refund_status VARCHAR(20) DEFAULT 'PENDING', -- PENDING, PROCESSED, FAILED

    CONSTRAINT check_percentages CHECK (refund_percentage BETWEEN 0 AND 100)
);

CREATE INDEX idx_cancellations_user ON cancellations(user_id);
CREATE INDEX idx_cancellations_ticket ON cancellations(ticket_id);
```

Pool_Tickets Table

```sql
```

```sql
CREATE TABLE pool_tickets (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    ticket_id UUID NOT NULL REFERENCES tickets(id) ON DELETE CASCADE,
    event_id UUID NOT NULL REFERENCES events(id) ON DELETE CASCADE,
    original_price DECIMAL(10, 2) NOT NULL,
    pool_price DECIMAL(10, 2) NOT NULL, -- Always equals original_price
    added_to_pool_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    nominated_user_id UUID REFERENCES users(id),
    nomination_expires_at TIMESTAMP,
    status VARCHAR(20) DEFAULT 'AVAILABLE', -- AVAILABLE, NOMINATED, CLAIMED, EXPIRE
    claimed_by UUID REFERENCES users(id),
    claimed_at TIMESTAMP,

    CONSTRAINT check_pool_price CHECK (pool_price = original_price),
    CONSTRAINT unique_ticket_in_pool UNIQUE (ticket_id)
);

CREATE INDEX idx_pool_tickets_status ON pool_tickets(status);
CREATE INDEX idx_pool_tickets_event ON pool_tickets(event_id);
CREATE INDEX idx_pool_tickets_nominated ON pool_tickets(nominated_user_id);
```

Payments Table

```sql
CREATE TABLE payments (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id),
    ticket_id UUID REFERENCES tickets(id),
    amount DECIMAL(10, 2) NOT NULL,
    payment_type VARCHAR(20) NOT NULL, -- PURCHASE, REFUND
    payment_method VARCHAR(50), -- CARD, UPI, NET_BANKING
    payment_gateway_id VARCHAR(255), -- Stripe/Razorpay transaction ID
    status VARCHAR(20) DEFAULT 'PENDING', -- PENDING, SUCCESS, FAILED
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_payments_user ON payments(user_id);
CREATE INDEX idx_payments_status ON payments(status);
```

Audit_Logs Table

```sql
```

```sql
CREATE TABLE audit_logs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id),
    action VARCHAR(100) NOT NULL, -- LOGIN, PURCHASE, TRANSFER, CANCEL, CLAIM
    entity_type VARCHAR(50), -- USER, TICKET, EVENT, POOL
    entity_id UUID,
    details JSONB, -- Store additional context
    ip_address VARCHAR(45),
    user_agent TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_audit_logs_user ON audit_logs(user_id);
CREATE INDEX idx_audit_logs_action ON audit_logs(action);
CREATE INDEX idx_audit_logs_created ON audit_logs(created_at);
```

🔐 Security Implementation (Spring Security 7)

Authentication Flow

1. User registers → email/phone verification sent
2. User logs in → JWT access token issued (valid 15 mins) + refresh token (valid 7 days)
3. Access token includes: user_id, email, roles, expiry
4. All protected API requests require valid JWT in `Authorization: Bearer <token>` header
5. Refresh token endpoint to get new access token without re-login
6. Token stored in HttpOnly cookie (web) or secure storage (mobile)

Authorization Levels

- **USER:** Can buy, view, transfer, cancel tickets; manage trusted circle
- **ADMIN:** Can create events, manage users, view analytics, handle disputes
- **VENUE_SCANNER:** Can verify tickets at entry (limited scope)
- **SUPPORT:** Can view tickets, help with issues, but can't transfer/cancel

Key Security Measures

- Password hashing with BCrypt (cost factor 12)

- JWT tokens with RSA-256 signing
- Rate limiting on authentication endpoints (5 attempts per 15 mins)
- HTTPS only in production (TLS 1.3)
- Input validation on all endpoints (Bean Validation)
- SQL injection prevention (JPA/Hibernate parameterized queries)
- XSS protection headers (Content-Security-Policy, X-Content-Type-Options)
- CSRF protection for state-changing operations
- Helmet.js equivalent security headers via Spring Security
- Redis-based session management for scalability

🎫 Dynamic QR Code Implementation (Leveraging Java 25)

Strategy with Virtual Threads

```java
```

```
// QR Code JSON structure
{
  "ticketId": "uuid",
  "eventId": "uuid",
  "timestamp": "epoch_milliseconds",
  "signature": "HMAC-SHA256(ticketId + eventId + timestamp, secret_key)"
}

// QR regenerates every 60 seconds using virtual threads
@Service
public class QRCodeService {

    @Async("virtualThreadExecutor") // Uses Java 25 virtual threads
    public CompletableFuture<String> generateDynamicQR(String ticketId) {
        // Runs on lightweight virtual thread
        String qrData = buildQRData(ticketId);
        String qrCodeImage = QRCodeGenerator.generate(qrData);

        // Cache in Redis with 90-second TTL
        redisTemplate.opsForValue().set(
            "qr:" + ticketId,
            qrCodeImage,
            90,
            TimeUnit.SECONDS
        );

        return CompletableFuture.completedFuture(qrCodeImage);
    }

    // Scanner validates using pattern matching (Java 25)
    public ValidationResult validateQR(String qrData) {
        return switch (parseQR(qrData)) {
            case ValidQR(var ticketId, var timestamp, var signature)
                when isSignatureValid(ticketId, timestamp, signature)
                && isTimestampRecent(timestamp, 90)
                -> new ValidationResult(true, ticketId);
            case InvalidQR(var reason)
                -> new ValidationResult(false, reason);
            default
                -> new ValidationResult(false, "Unknown QR format");
        };
    }
}
```

Benefits

- Screenshots become invalid after 90 seconds
- Prevents ticket duplication and screenshots
- Each scan validates signature cryptographically
- Original ticket seed stored in database (never changes)
- Dynamic QR stored in Redis for fast retrieval
- Virtual threads handle thousands of QR generations concurrently

Redis QR Cache Structure

```
Key: "qr:{ticketId}"
Value: Base64-encoded QR code image
TTL: 90 seconds
```

🚀 MVP Development Phases (8-10 Week Timeline)

Phase 1: Foundation (Week 1-2)

- [ ] Set up Spring Boot 4 + Java 25 project with Maven
- [ ] Configure PostgreSQL 17 + Redis 7 (Docker Compose)
- [ ] Implement user registration/login with Spring Security 7
- [ ] JWT token generation and validation
- [ ] Basic API structure with proper error handling
- [ ] Swagger/OpenAPI documentation setup
- [ ] Database migrations with Flyway or Liquibase
- [ ] Configure virtual threads executor

Phase 2: Core Ticketing (Week 3-4)

- [ ] Event creation (admin only)
- [ ] Event listing and filtering
- [ ] Ticket purchase flow with validation
- [ ] Payment gateway integration (Stripe/Razorpay test mode)
- [ ] Dynamic QR code generation with Redis caching
- [ ] "My Tickets" dashboard
- [ ] Purchase limits and inventory management

Phase 3: Transfer System (Week 5-6)

- [ ] Trusted circle CRUD operations

- [ ] Trusted circle transfer logic (instant, no restrictions)
- [ ] Controlled transfer logic with time/count validation
- [ ] Transfer verification flow
- [ ] Transfer history tracking
- [ ] Pattern detection for suspicious transfers
- [ ] Email notifications for transfers

Phase 4: Cancellation & Pool (Week 7-8)

- [ ] Time-based cancellation logic with dynamic refund calculation
- [ ] Refund processing integration
- [ ] Pool ticket creation on cancellation
- [ ] Pool marketplace UI/API
- [ ] Ticket claiming system (with virtual threads for concurrency)
- [ ] Nomination system (15-min priority window)
- [ ] WebSocket or SSE for real-time pool updates
- [ ] Email/SMS notifications for nominations

Phase 5: Anti-Fraud & Polish (Week 9-10)

- [ ] Pattern detection algorithms (velocity checks, suspicious transfers)
- [ ] Account flagging system with severity levels
- [ ] Admin dashboard for reviewing violations
- [ ] Venue scanner simulation/API
- [ ] Entry validation and ticket marking
- [ ] Comprehensive unit and integration tests
- [ ] API documentation finalization
- [ ] Performance testing and optimization
- [ ] Security audit and penetration testing
- [ ] Deployment configuration (Docker, CI/CD)

🧪 Testing Strategy

Unit Tests (JUnit 5 + Mockito)

- Service layer business logic
- Pricing and refund calculations
- Transfer validation rules
- QR code generation and validation
- Pattern matching logic

Integration Tests (Spring Boot Test + TestContainers)

- API endpoints (all CRUD operations)
- Database operations with real PostgreSQL
- Redis caching behavior
- Payment gateway integration (test mode)
- Authentication and authorization flows

Performance Tests (JMeter or Gatling)

- Concurrent pool ticket claims (1000+ simultaneous users)
- QR code generation under load
- API response times under stress
- Database query performance

Key Test Scenarios

- ✅ User can't transfer ticket more than max limit
- ✅ Price cap is enforced (no markup possible)
- ✅ QR codes expire correctly after 90 seconds
- ✅ Pool claims are atomic (no double-claims via database constraints)
- ✅ Refund percentages calculate correctly based on time
- ✅ Trusted circle transfers work instantly
- ✅ Non-circle transfers enforce 48-hour rule
- ✅ Nomination window expires after 15 minutes
- ✅ Virtual threads handle 10,000+ concurrent operations
- ✅ Pattern detection flags suspicious accounts
- ✅ Used tickets can't be scanned again

📊 MVP Success Metrics

Technical KPIs

- [ ] All APIs respond in <500ms (p95)
- [ ] Zero duplicate ticket claims (enforced by DB constraints + Redis locks)
- [ ] 99.5%+ uptime
- [ ] QR validation works 100% of time
- [ ] Support 1000+ concurrent pool claims without degradation

- [ ] Virtual threads reduce thread count by 90%+ vs traditional threads

Business KPIs

- [ ] Users can complete purchase in <2 minutes
- [ ] Transfer success rate >95%
- [ ] Pool tickets claimed within 24 hours (80%+ claim rate)
- [ ] Zero successful scalping attempts detected
- [ ] User satisfaction score >4/5
- [ ] Event organizer adoption rate >20% in pilot

🎯 Post-MVP: Migration to Microservices

Once MVP is validated (3-6 months), break monolith into services:

1. **User Service** - Auth, profiles, trusted circles, verification
2. **Event Service** - Event management, capacity tracking
3. **Ticket Service** - Ownership, QR generation, status management
4. **Transfer Service** - All transfer logic and validation
5. **Pool Service** - Marketplace, claims, nominations
6. **Payment Service** - Transactions, refunds, gateway integration
7. **Fraud Service** - Pattern detection, account flagging
8. **Notification Service** - Emails, SMS, push notifications
9. **Scanner Service** - Venue entry validation
10. **Analytics Service** - Metrics, reporting, insights

Communication: REST APIs + RabbitMQ/Kafka for async events **Service Discovery:** Spring Cloud Eureka or Consul **API Gateway:** Spring Cloud Gateway **Config Management:** Spring Cloud Config Server

---

💡 pom.xml for Java 25 + Spring Boot 4

```xml
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>4.0.0</version> <!-- Spring Boot 4 -->
        <relativePath/>
    </parent>

    <groupId>com.antiscalping</groupId>
    <artifactId>ticket-platform</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>Anti-Scalping Ticket Platform</name>
    <description>Open-source platform to eliminate ticket scalping</description>

    <properties>
        <java.version>25</java.version>
        <maven.compiler.source>25</maven.compiler.source>
        <maven.compiler.target>25</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- Spring Boot Core Starters -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-websocket</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>

    <!-- Database -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Database Migration -->
    <dependency>
        <groupId>org.flywaydb</groupId>
        <artifactId>flyway-core</artifactId>
    </dependency>

    <dependency>
        <groupId>org.flywaydb</groupId>
        <artifactId>flyway-database-postgresql</artifactId>
    </dependency>

    <!-- JWT -->
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>0.12.5</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>0.12.5</version>
        <scope>runtime</scope>
```

```
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.12.5</version>
    <scope>runtime</scope>
</dependency>

<!-- QR Code Generation -->
<dependency>
    <groupId>com.google.zxing</groupId>
    <artifactId>core</artifactId>
    <version>3.5.3</version>
</dependency>
<dependency>
    <groupId>com.google.zxing</groupId>
    <artifactId>javase</artifactId>
    <version>3.5.3</version>
</dependency>

<!-- Lombok (Code Generation) -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>

<!-- API Documentation (OpenAPI 3 / Swagger) -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.6.0</version>
</dependency>

<!-- Monitoring & Metrics -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>

<!-- Payment Gateway (Choose one) -->
<!-- Stripe SDK -->
```

```
<dependency>
    <groupId>com.stripe</groupId>
    <artifactId>stripe-java</artifactId>
    <version>26.13.0</version>
</dependency>

<!-- OR Razorpay SDK (for India)
<dependency>
    <groupId>com.razorpay</groupId>
    <artifactId>razorpay-java</artifactId>
    <version>1.4.6</version>
</dependency>
-->

<!-- Testing -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>

<!-- TestContainers for Integration Tests -->
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.20.4</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>1.20.4</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.20.4</version>
    <scope>test</scope>
```

```xml
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>

            <!-- Maven Compiler Plugin for Java 25 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.13.0</version>
                <configuration>
                    <source>25</source>
                    <target>25</target>
                    <compilerArgs>
                        <arg>--enable-preview</arg> <!-- If using preview features
                    </compilerArgs>
                </configuration>
            </plugin>

            <!-- Flyway Migration Plugin -->
            <plugin>
                <groupId>org.flywaydb</groupId>
                <artifactId>flyway-maven-plugin</artifactId>
                <version>10.21.0</version>
            </plugin>
        </plugins>
    </build>
</project>
```

🐬 Docker Compose for Development

```yaml
```

```yaml
version: '3.8'

services:
  postgres:
    image: postgres:17-alpine
    container_name: ticket-platform-db
    environment:
      POSTGRES_DB: ticket_platform
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secure_password_here
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - ticket-network

  redis:
    image: redis:7-alpine
    container_name: ticket-platform-redis
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    networks:
      - ticket-network

  # Optional: RabbitMQ for async messaging
  rabbitmq:
    image: rabbitmq:3.13-management-alpine
    container_name: ticket-platform-rabbitmq
    environment:
      RABBITMQ_DEFAULT_USER: admin
      RABBITMQ_DEFAULT_PASS: secure_password_here
    ports:
      - "5672:5672"   # AMQP port
      - "15672:15672" # Management UI
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq
    networks:
      - ticket-network

volumes:
  postgres_data:
  redis_data:
  rabbitmq_data:
```

```
networks:
  ticket-network:
    driver: bridge
```

Start services:

```bash
docker-compose up -d
```

🔧 application.yml Configuration

```yaml
```

```yaml
spring:
  application:
    name: anti-scalping-ticket-platform

  datasource:
    url: jdbc:postgresql://localhost:5432/ticket_platform
    username: admin
    password: secure_password_here
    driver-class-name: org.postgresql.Driver

  jpa:
    hibernate:
      ddl-auto: validate # Use Flyway for migrations
    show-sql: false
    properties:
      hibernate:
        format_sql: true
        dialect: org.hibernate.dialect.PostgreSQLDialect

  data:
    redis:
      host: localhost
      port: 6379
      timeout: 2000ms

  security:
    jwt:
      secret-key: ${JWT_SECRET_KEY:your-256-bit-secret-key-here-change-in-productio
      expiration: 900000 # 15 minutes in milliseconds
      refresh-expiration: 604800000 # 7 days

  mail:
    host: smtp.gmail.com
    port: 587
    username: ${MAIL_USERNAME}
    password: ${MAIL_PASSWORD}
    properties:
      mail:
        smtp:
          auth: true
          starttls:
            enable: true

# Payment Gateway Configuration
payment:
  stripe:
```

```yaml
    api-key: ${STRIPE_API_KEY}
    webhook-secret: ${STRIPE_WEBHOOK_SECRET}
  # razorpay:
  #   key-id: ${RAZORPAY_KEY_ID}
  #   key-secret: ${RAZORPAY_KEY_SECRET}

# Application-Specific Configuration
ticket-platform:
  max-purchase-per-event: 4
  max-trusted-circle-size: 5
  max-ticket-transfers: 2
  controlled-transfer-hours-limit: 48
  pool-nomination-window-minutes: 15
  qr-code-validity-seconds: 90

  refund-tiers:
    - hours: 24
      percentage: 90
    - hours: 12
      percentage: 70
    - hours: 6
      percentage: 50
    - hours: 0
      percentage: 25

# Actuator Configuration
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,prometheus
  metrics:
    export:
      prometheus:
        enabled: true

# Logging
logging:
  level:
    root: INFO
    com.antiscalping.tickets: DEBUG
    org.springframework.security: DEBUG
  pattern:
    console: "%d{yyyy-MM-dd HH:mm:ss} - %msg%n"
```

🎨 Leveraging Java 25 Features - Code Examples

1. Virtual Threads for Concurrent Pool Claims

```java
```

```java
@Configuration
public class AsyncConfig {

    @Bean(name = "virtualThreadExecutor")
    public Executor virtualThreadExecutor() {
        return Executors.newVirtualThreadPerTaskExecutor();
    }
}

@Service
public class PoolService {

    @Async("virtualThreadExecutor")
    public CompletableFuture<PoolClaimResult> claimTicket(
        String poolTicketId,
        String userId
    ) {
        // Each claim runs in a lightweight virtual thread
        // Can handle 10,000+ concurrent claims efficiently

        try {
            // Atomic claim with Redis lock to prevent double-claims
            String lockKey = "claim-lock:" + poolTicketId;
            Boolean acquired = redisTemplate.opsForValue()
                .setIfAbsent(lockKey, userId, 5, TimeUnit.SECONDS);

            if (Boolean.FALSE.equals(acquired)) {
                return CompletableFuture.completedFuture(
                    new PoolClaimResult(false, "Already claimed")
                );
            }

            // Process claim in database
            PoolTicket poolTicket = poolTicketRepository
                .findById(UUID.fromString(poolTicketId))
                .orElseThrow();

            if (poolTicket.getStatus() != PoolStatus.AVAILABLE) {
                return CompletableFuture.completedFuture(
                    new PoolClaimResult(false, "No longer available")
                );
            }

            // Transfer ticket to new owner
            Ticket ticket = poolTicket.getTicket();
            ticket.setOwnerId(UUID.fromString(userId));
```

```
            ticket.setStatus(TicketStatus.ACTIVE);
            ticketRepository.save(ticket);

            poolTicket.setStatus(PoolStatus.CLAIMED);
            poolTicket.setClaimedBy(UUID.fromString(userId));
            poolTicket.setClaimedAt(LocalDateTime.now());
            poolTicketRepository.save(poolTicket);

            return CompletableFuture.completedFuture(
                new PoolClaimResult(true, "Claimed successfully")
            );

        } finally {
            redisTemplate.delete("claim-lock:" + poolTicketId);
        }
    }
}
```

2. Pattern Matching for Transfer Validation

java

```java
@Service
public class TransferService {

    public TransferValidation validateTransfer(
        Ticket ticket,
        User from,
        User to,
        TransferType type
    ) {
        // Use Java 25 pattern matching with guards
        return switch (type) {
            case TRUSTED_CIRCLE when isTrustedCircle(from, to) ->
                new TransferValidation(true, "Trusted circle transfer");

            case CONTROLLED when ticket.getTransferCount() >= ticket.getMaxTransfers
                new TransferValidation(false, "Transfer limit exceeded");

            case CONTROLLED when !hasEnoughTimeBeforeEvent(ticket, 48) ->
                new TransferValidation(false, "Must transfer 48+ hours before event"

            case CONTROLLED ->
                new TransferValidation(true, "Controlled transfer allowed");

            default ->
                new TransferValidation(false, "Invalid transfer type");
        };
    }

    private boolean hasEnoughTimeBeforeEvent(Ticket ticket, int requiredHours) {
        Event event = eventRepository.findById(ticket.getEventId()).orElseThrow();
        long hoursUntilEvent = ChronoUnit.HOURS.between(
            LocalDateTime.now(),
            event.getEventDate()
        );
        return hoursUntilEvent >= requiredHours;
    }
}
```

3. Record Patterns for DTOs

```java
```

```java
// Immutable request/response DTOs using Records
public record TicketPurchaseRequest(
    @NotNull UUID eventId,
    @Min(1) @Max(4) int quantity,
    @NotBlank String paymentMethodId
) {}

public record TransferRequest(
    @NotNull UUID ticketId,
    @NotNull UUID toUserId,
    @NotNull TransferType transferType,
    String reason
) {}

public record PoolTicketResponse(
    UUID id,
    String eventName,
    LocalDateTime eventDate,
    BigDecimal price,
    LocalDateTime addedToPool,
    boolean isNominated
) {}

// Pattern matching with records
public PaymentResult processPayment(PaymentRequest request) {
    return switch (request) {
        case PaymentRequest(var amount, var method, var userId)
            when amount.compareTo(BigDecimal.ZERO) > 0 ->
            processValidPayment(amount, method, userId);

        case PaymentRequest(var amount, _, _)
            when amount.compareTo(BigDecimal.ZERO) <= 0 ->
            new PaymentResult(false, "Invalid amount");

        default ->
            new PaymentResult(false, "Invalid payment request");
    };
}
```

4. String Templates for Dynamic Queries (if using JPQL)

```java
```

```
// Note: String templates are a preview feature in Java 25
// Enable with --enable-preview flag

public List<Event> searchEvents(String keyword, LocalDateTime after) {
    String query = STR."""
        SELECT e FROM Event e
        WHERE e.name LIKE '%\{keyword}%'
        AND e.eventDate > :afterDate
        AND e.status = 'UPCOMING'
        ORDER BY e.eventDate ASC
        """;

    return entityManager.createQuery(query, Event.class)
        .setParameter("afterDate", after)
        .getResultList();
}
```

💡 Quick Start Commands

bash

```
# 1. Create project using Spring Initializr (or IDE)
spring init \
  --dependencies=web,data-jpa,security,data-redis,validation,actuator \
  --type=maven-project \
  --java-version=25 \
  --boot-version=4.0.0 \
  --artifactId=ticket-platform \
  --package-name=com.antiscalping.tickets \
  anti-scalping-tickets

cd anti-scalping-tickets

# 2. Start PostgreSQL & Redis via Docker
docker-compose up -d

# 3. Verify services are running
docker ps

# 4. Run the application
./mvnw clean spring-boot:run

# 5. Access Swagger UI (API documentation)
# Open browser: http://localhost:8080/swagger-ui.html

# 6. Access Actuator endpoints
# http://localhost:8080/actuator/health
# http://localhost:8080/actuator/metrics

# 7. Run tests
./mvnw test

# 8. Build for production
./mvnw clean package -DskipTests
```

🎨 MVP Frontend (Optional - Build After Backend Works)

For MVP testing, you can:

1. **Use Swagger UI** for API testing initially (built-in)
2. **Build simple React/Vue SPA** later for user-facing features
3. **Use Thymeleaf** for server-side rendering (simpler, but less modern)
4. **Mobile apps** come after web validation (React Native or Flutter)

Recommended Approach: Build backend first, test with Swagger/Postman, then add frontend once APIs are stable.

📋 MVP Exclusions (Add in Phase 2)

Features intentionally left out of MVP:

- ❌ Seated events (Phase 2 feature)
- ❌ Advanced analytics dashboard (basic metrics only)
- ❌ Rich email templates (plain text for MVP)
- ❌ SMS notifications (email only for MVP)
- ❌ Advanced bot detection (simple rate limiting for MVP)
- ❌ Mobile apps (web-first approach)
- ❌ Multiple venue integrations (simulate scanner for MVP)
- ❌ Lottery system for pool claims (FCFS is simpler)
- ❌ Blockchain audit trail (overkill for MVP)
- ❌ Multi-language support (English only for MVP)
- ❌ Social media sharing (conflicts with anti-scalping goals)

✅ Ready to Build Checklist

Before you start coding:

- [ ] Java 25 JDK installed and configured
- [ ] Maven 3.9+ installed
- [ ] PostgreSQL 17 running (Docker or local)
- [ ] Redis 7 running (Docker or local)
- [ ] IDE set up (IntelliJ IDEA recommended for Java 25 support)
- [ ] Git repository created (GitHub/GitLab)
- [ ] Payment gateway test account (Stripe or Razorpay)
- [ ] Database schema designed (see above)
- [ ] API endpoints mapped out
- [ ] Environment variables configured (.env file)
- [ ] Docker installed for services

🎯 Final MVP Goal

Build a working system where:

1. ✅ Users can register, verify email, and log in securely
2. ✅ Users can buy tickets for unseated events with purchase limits
3. ✅ Users can add trusted friends and transfer tickets instantly to them
4. ✅ Users can transfer to others with 48-hour rule and max 2 transfers
5. ✅ Users can cancel tickets and get time-based refunds (90% to 25%)
6. ✅ Cancelled tickets automatically go to pool marketplace
7. ✅ Users can nominate someone for 15-minute priority claim
8. ✅ Users can claim tickets from pool at retail price only
9. ✅ All tickets use dynamic QR codes that refresh every 60 seconds
10. ✅ Pattern detection flags suspicious transfer behavior
11. ✅ Virtual threads handle 1000+ concurrent pool claims
12. ✅ Admin can create events and manage users

Development Timeline: 8-10 weeks for solo developer, 4-6 weeks for small team

After MVP works:

1. Get 1-2 small local events to pilot the system
2. Gather feedback from users and organizers
3. Fix bugs and optimize based on real usage
4. Plan Phase 2 (seated events, mobile apps)
5. Begin migration to microservices architecture
6. Prepare for open-source launch

🚀 You're All Set!

This is your complete, updated blueprint with Java 25, Spring Boot 4, and Spring 7. The tech stack is modern, performant, and future-proof.

Start Building Today:

1. Set up your development environment
2. Initialize the Spring Boot project with Maven

3. Start with user authentication (hardest part first)
4. Build events, then tickets, then transfers, then pool
5. Test each feature thoroughly before moving to the next
6. Keep the code clean, documented, and testable

Good luck with your MVP! This has real potential to disrupt the ticketing industry. 🎫✨