Nexgate Platform - Check-in System Architecture

Table of Contents

1. System Overview
2. Architecture Principles
3. System Components
4. Ticket Generation Flow
5. Scanner Registration Flow
6. Ticket Validation Flow
7. Offline Mode Architecture
8. Security Model
9. Data Models
10. API Specifications
11. Database Schema
12. Deployment Architecture

System Overview

What is the Check-in System?

The Check-in System is a critical component of the Nexgate platform that handles secure, scalable ticket validation for events. It enables event organizers to verify attendee tickets at entry gates using mobile scanner devices, with the capability to work both online and offline.

Key Features

- **Cryptographic Security**: Uses RSA-signed JWT tokens to prevent ticket forgery
- **Offline Capability**: Scanners can validate tickets without internet connectivity

- **Multi-Gate Support**: Coordinate validation across multiple entry points
- **Real-time Validation**: Immediate duplicate detection when online
- **Scanner Management**: Secure device registration and revocation
- **Audit Trail**: Complete tracking of all scan activities

Inspiration

The system is inspired by electrical meter voucher systems commonly used in Tanzania and other African countries, where vouchers must be validated offline after purchase, with reconciliation happening later when connectivity is restored.

Architecture Principles

1. Security First

- All tickets are cryptographically signed
- Cannot be forged without the server's private key
- Scanner devices only receive public keys for verification

2. Offline-First Design

- Scanners must function without network connectivity
- Local validation using JWT signature verification
- Queue-based synchronization when connectivity returns

3. Zero-Trust Scanner Model

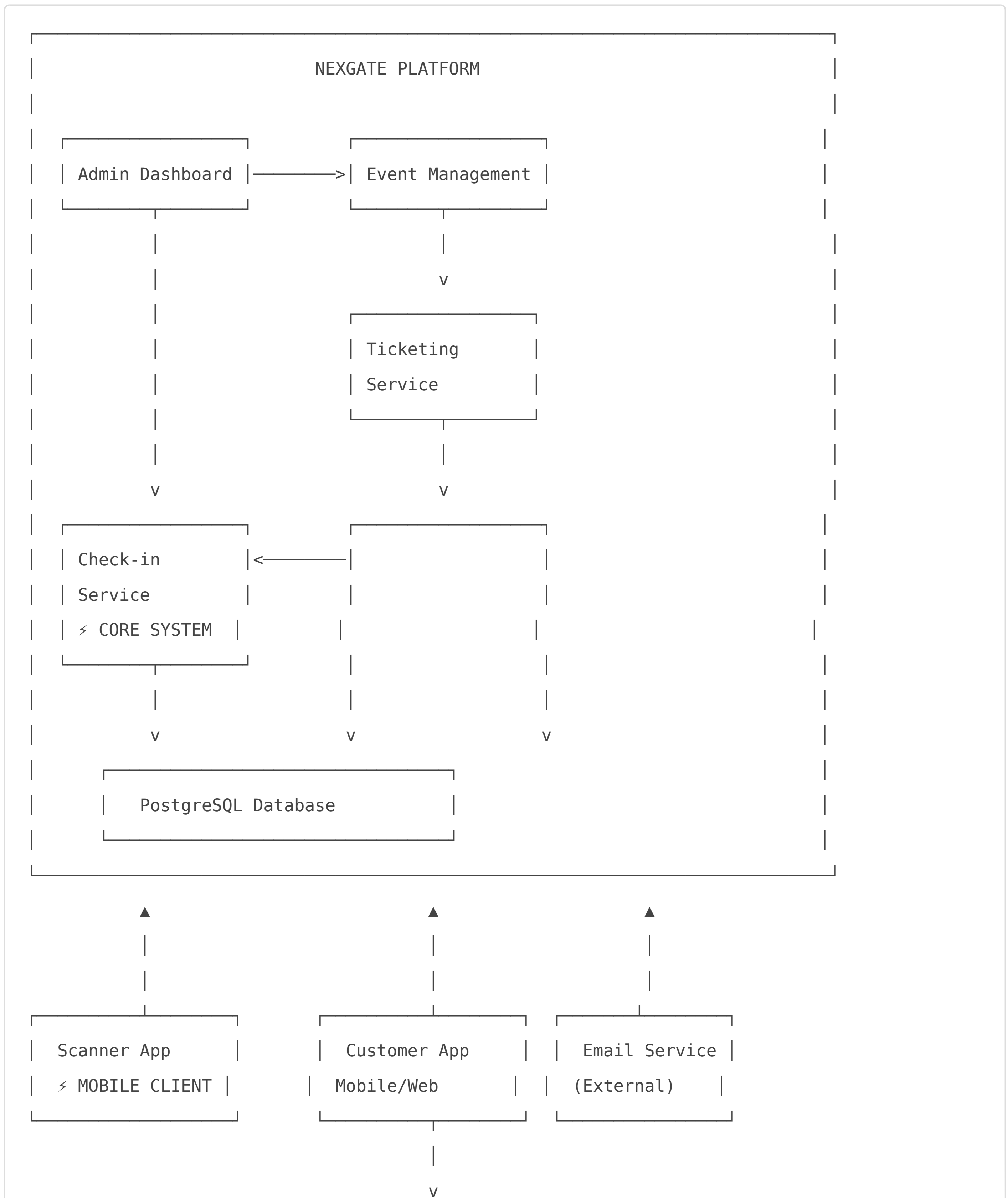
- Each scanner is individually registered and can be revoked
- Scanners receive time-limited registration tokens
- All scanner actions are logged and auditable

4. Scalability

- Stateless ticket validation (JWT-based)
- No database queries required for offline validation
- Server handles only registration and synchronization

System Components

Component Diagram



```
                          ┌────────────────────┐
                          │  Payment Gateway   │
                          │  (External)        │
                          └────────────────────┘


Legend:
  ──────>  Data Flow
  ⚡       Critical Component
```

Core Components

1. Check-in Service (Spring Boot Backend)

Responsibilities:

- Generate and manage scanner registration tokens
- Issue scanner credentials
- Generate RSA key pairs for ticket signing
- Create JWT-based tickets with QR codes
- Handle online ticket validation requests
- Receive and process scan logs from scanners
- Manage scanner settings and configurations
- Maintain scan history and analytics

Technology Stack:

- Spring Boot 3.x
- Spring Security
- PostgreSQL
- Redis (for caching and rate limiting)
- JWT (io.jsonwebtoken library)
- ZXing (QR code generation)

2. Scanner Mobile App (Android)

Responsibilities:

- Register scanner device using QR code
- Store scanner credentials and server public key
- Scan ticket QR codes
- Validate tickets offline using JWT verification
- Validate tickets online when connected
- Queue scan logs for synchronization
- Sync with server periodically

- Display scan history and statistics

Technology Stack:

- Android (Kotlin)
- Room Database (local storage)
- WorkManager (background sync)
- ZXing (QR code scanning)
- JWT library for validation
- Retrofit (API communication)

3. Admin Dashboard (Web)

Responsibilities:

- Generate scanner registration QR codes
- View and manage registered scanners
- Revoke scanner access
- Configure scanner settings
- View scan analytics and reports
- Monitor real-time scanning activity
- Export scan data

Technology Stack:

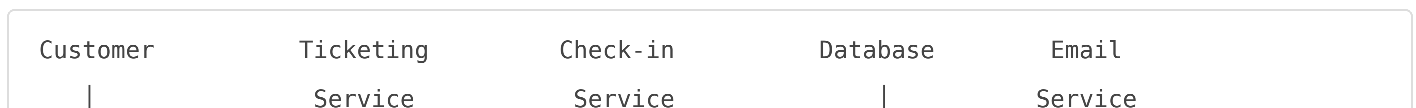
- React.js / Vue.js
- Chart.js (analytics)
- WebSocket (real-time updates)

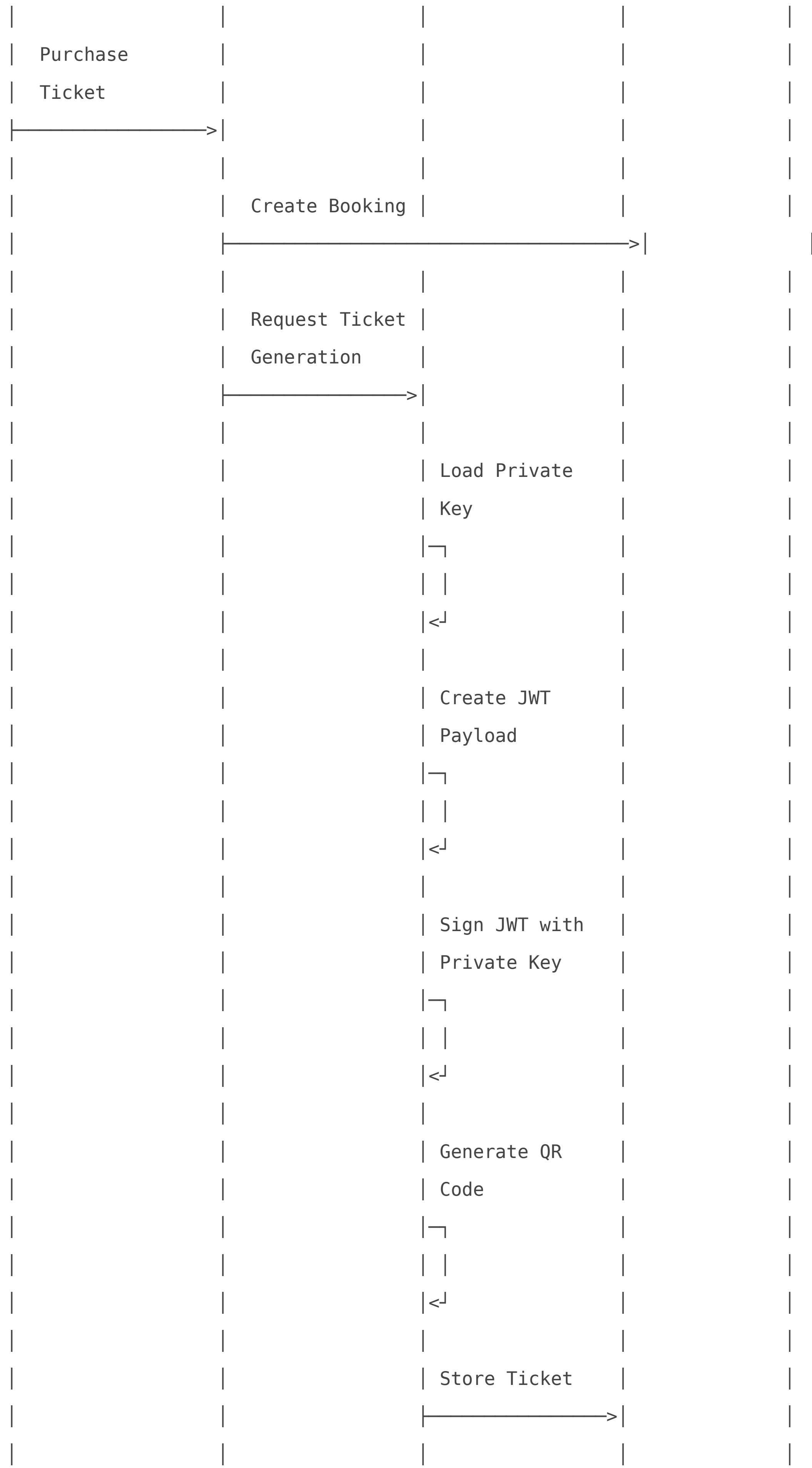
Ticket Generation Flow

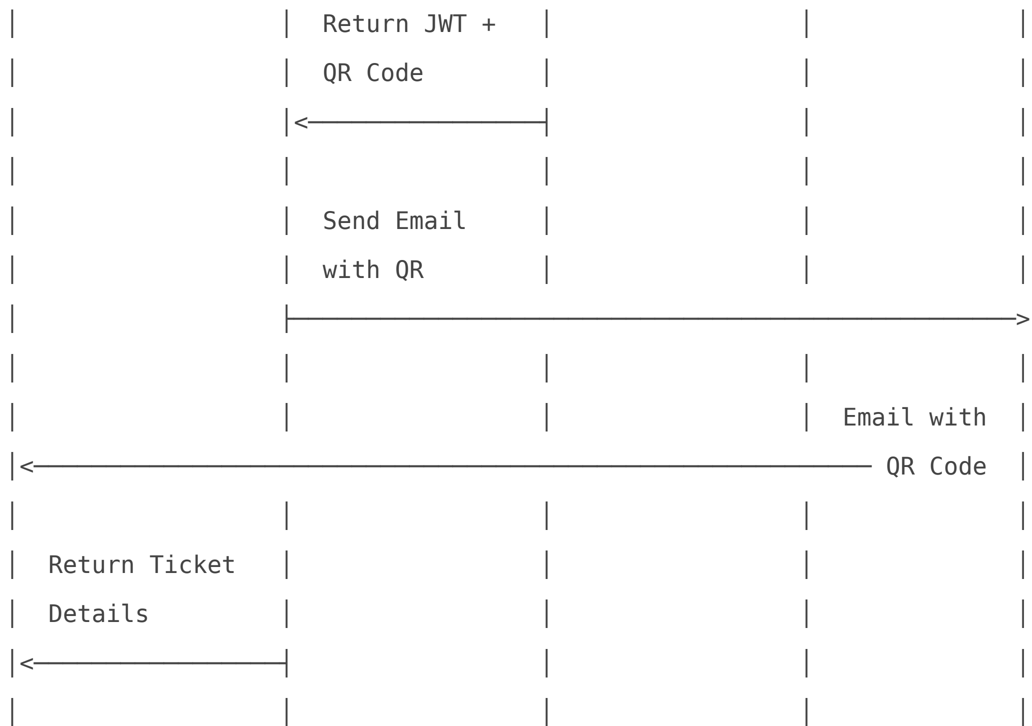
Overview

When a customer purchases a ticket, the system generates a cryptographically signed JWT token that is embedded in a QR code. This QR code serves as the ticket that the customer presents at the event gate.

Detailed Flow




Purchase
Ticket
Create Booking
Request Ticket
Generation
Load Private
Key
Create JWT
Payload
Sign JWT with
Private Key
Generate QR
Code
Store Ticket




```
Legend:
  ───>   Synchronous request
  ──>    Response
  ┐ │    Internal processing
```

Step-by-Step Process

Step 1: Customer Completes Purchase

```
Customer → Ticketing Service
- Selects event and ticket type
- Completes payment
- Receives booking confirmation
```

Step 2: Ticketing Service Requests Ticket Generation

```
Ticketing Service → Check-in Service
POST /api/tickets/generate
{
  "bookingId": "booking-uuid",
  "eventId": "event-uuid",
  "attendeeName": "John Doe",
  "attendeeEmail": "john@example.com",
  "ticketType": "VIP",
```

```
  "validFrom": "2025-12-01T18:00:00Z",
  "validUntil": "2025-12-01T23:59:59Z"
}
```

Step 3: Check-in Service Creates JWT

JWT Header:

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

JWT Payload:

```
{
  "ticketId": "ticket-uuid-123",
  "bookingId": "booking-uuid",
  "eventId": "event-uuid",
  "eventName": "Tech Conference 2025",
  "attendeeName": "John Doe",
  "attendeeEmail": "john@example.com",
  "ticketType": "VIP",
  "seatNumber": "A-12",
  "iat": 1701234567,
  "exp": 1703826567,
  "nbf": 1701234567
}
```

Signing Process:

```
1. Encode Header as Base64URL
2. Encode Payload as Base64URL
3. Create signature:
   SHA256withRSA(base64(header) + "." + base64(payload), PRIVATE_KEY)
4. Final JWT = header.payload.signature
```

Step 4: Generate QR Code

```
1. Take complete JWT string
2. Generate QR code image (300x300 pixels)
```

```
3. Encode as Base64 string
4. Store in ticket record
```

Step 5: Store and Distribute

```
Database Record:
- ticket_id
- booking_id
- event_id
- jwt_token (full JWT string)
- qr_code_base64
- status (ACTIVE, SCANNED, CANCELLED)
- created_at

Distribution:
- Email to customer with QR code image
- Available in mobile app
- Printable PDF option
```

Ticket JWT Example


```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0aWNrZXRJZCI6ImFiYy0xMjMiLCJldmVudElkIjoiZXZlbnQtMDAxI
iwiYXR0ZW5kZWVOYW1lIjoiSm9obiBEb2UiLCJhdHRlbmRlZUVtYWlsIjoiam9obkBleGFtcGxlLmNvbSIsImlhdCI6MTc
wMTIzNDU2NywiZXhwIjoxNzAzODI2NTY3fQ.signature_here
```

Scanner Registration Flow

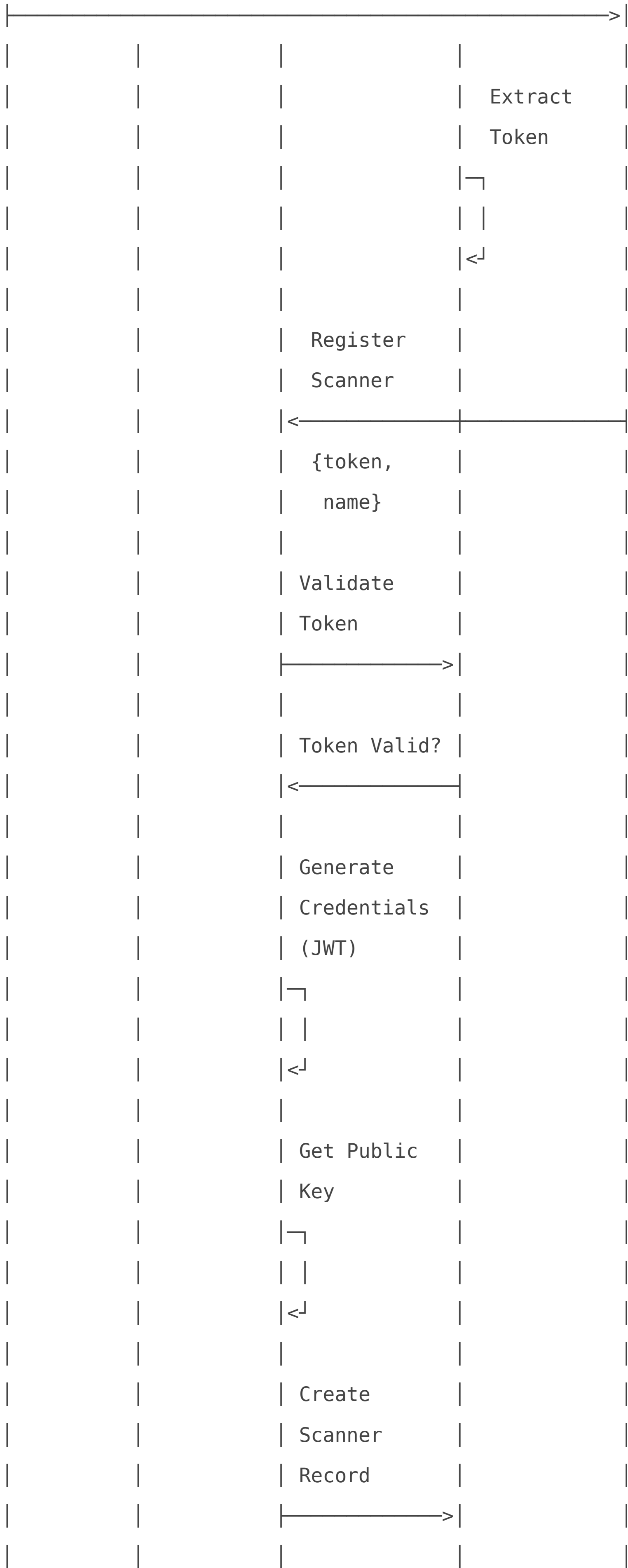
Overview

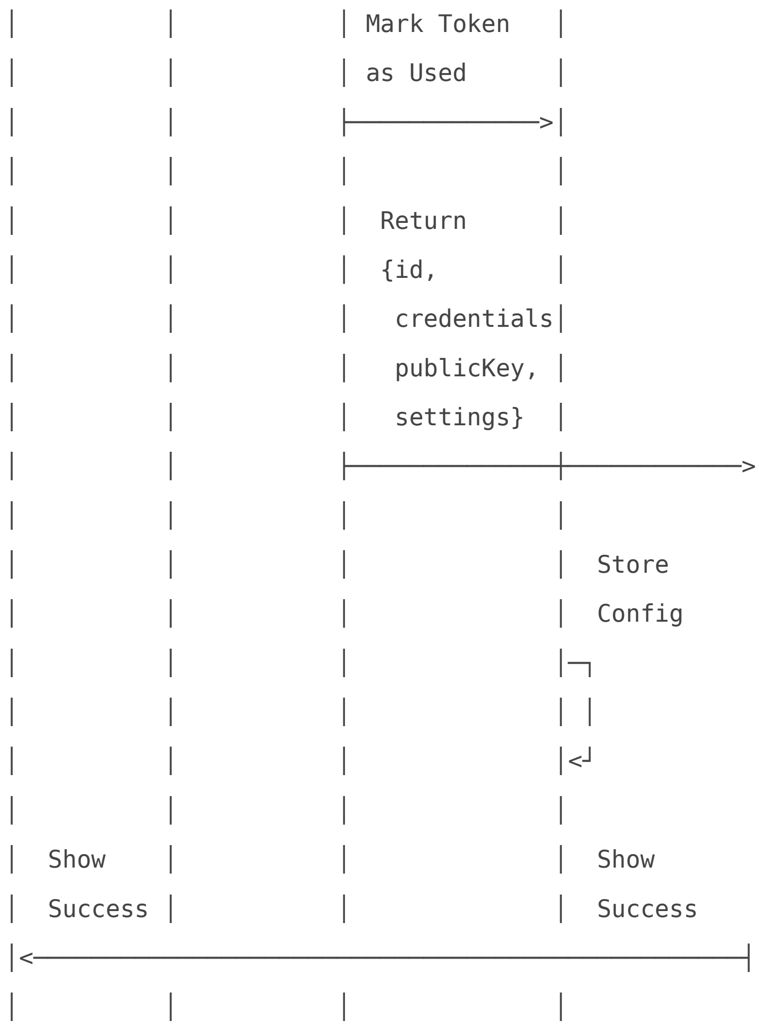
Before a scanner device can validate tickets, it must be registered with the system. This process is similar to WhatsApp's "Link Device" feature, using a time-limited QR code for secure pairing.

Detailed Flow

```
Admin    Dashboard    Check-in       Database      Scanner
                      Service                       App
```

```
     |           |            |              |             |
     | Click     |            |              |             |
     | "Add      |            |              |             |
     | Scanner"  |            |              |             |
     ├──────────>|            |              |             |
     |           |            |              |             |
     |           | Generate   |              |             |
     |           | Token      |              |             |
     |           ├───────────>|              |             |
     |           |            |              |             |
     |           |            | Create Token |             |
     |           |            | (UUID)       |             |
     |           |            | Set Expiry   |             |
     |           |            | (5 min)      |             |
     |           |            |─┐            |             |
     |           |            | |            |             |
     |           |            |<┘            |             |
     |           |            |              |             |
     |           |            | Store Token  |             |
     |           |            ├─────────────>|             |
     |           |            |              |             |
     |           |  Return    |              |             |
     |           |  Token     |              |             |
     |           |<───────────┤              |             |
     |           |            |              |             |
     |           | Generate   |              |             |
     |           | QR Code    |              |             |
     |           |─┐          |              |             |
     |           | |          |              |             |
     |           |<┘          |              |             |
     |           |            |              |             |
     |  Display  |            |              |             |
     |  QR Code  |            |              |             |
     |<──────────┤            |              |             |
     |           |            |              |             |
     |                                                     |
     |                 Scanner scans QR code               |
     |                                                     |
     |  Scan QR  |            |              |             |
```


Extract
Token
Register
Scanner
{token,
name}
Validate
Token
Token Valid?
Generate
Credentials
(JWT)
Get Public
Key
Create
Scanner
Record




```
Legend:
  ──>   Request/Response
  ┐ |   Internal processing
  ....  Scanner scans QR
```

Step-by-Step Process

Step 1: Admin Initiates Registration

```
Admin Dashboard:
1. Navigate to "Scanners" section
2. Click "Add New Scanner" button
3. Specify scanner details:
   - Scanner Name: "Gate A - Main Entrance"
   - Validity: 5 minutes (default)
   - Notes: Optional description
```

Step 2: Server Generates Registration Token

```
POST /api/registration-tokens/generate
Request:
{
  "validityMinutes": 5,
  "notes": "Gate A scanner for Main Entrance"
}

Server Process:
1. Generate UUID token
2. Calculate expiry time (now + 5 minutes)
3. Store in database:
   - token: "abc-123-xyz-789"
   - expires_at: "2025-11-29T10:15:00Z"
   - used: false
   - created_by: "admin@nexgate.com"

Response:
{
  "token": "abc-123-xyz-789",
  "qrCodeBase64": "data:image/png;base64,iVBORw0KG...",
  "expiresAt": "2025-11-29T10:15:00Z",
  "validityMinutes": 5
}
```

Step 3: Display QR Code

```
Admin Dashboard displays:
- Large QR code containing the token
- Expiry countdown timer
- Token details
- "Waiting for scanner to connect..." message
```

Step 4: Scanner Scans QR Code

```
Scanner App:
1. Open camera for QR scanning
2. Scan QR code displayed on admin dashboard
3. Extract token string: "abc-123-xyz-789"
4. Prompt user to confirm device name
```

Step 5: Scanner Sends Registration Request

```
POST /api/scanners/register
Request:
{
  "token": "abc-123-xyz-789",
  "deviceName": "Gate A Scanner",
  "deviceInfo": {
    "model": "Samsung Galaxy S21",
    "osVersion": "Android 14",
    "appVersion": "1.0.0"
  }
}
```

Step 6: Server Validates and Issues Credentials

```
Server Validation:
1. Find token in database
2. Check if token exists
3. Check if token.used == false
4. Check if token.expiresAt > now()

If valid:
1. Generate scanner credentials (JWT):
   {
     "scannerId": "scanner-uuid",
     "scannerName": "Gate A Scanner",
     "type": "scanner_credential",
     "iat": now,
     "exp": now + 1 year
   }

2. Sign with server private key

3. Create scanner record in database:
   - scanner_id: UUID
   - name: "Gate A Scanner"
   - credentials: JWT
   - status: ACTIVE
   - settings: default settings JSON
   - created_at: now
```

```
4. Mark token as used:
   - used: true
   - used_at: now
   - scanner_name: "Gate A Scanner"

Response:
{
  "scannerId": "scanner-uuid",
  "credentials": "eyJhbGc...scanner_jwt",
  "publicKey": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA...",
  "settings": {
    "offlineModeEnabled": false,
    "syncIntervalMinutes": 15,
    "maxOfflineHours": 24
  }
}
```

Step 7: Scanner Stores Configuration

```
Scanner App (Local Storage):
1. Save scanner ID
2. Save credentials JWT
3. Convert publicKey string to PublicKey object
4. Save public key
5. Save settings
6. Mark device as "registered"

Scanner is now ready to validate tickets!
```

Registration Token Lifecycle

```
stateDiagram-v2
    [*] --> Generated: Admin creates token
    Generated --> Active: Token created, timer starts
    Active --> Used: Scanner registers successfully
    Active --> Expired: Time runs out
    Used --> [*]: Token consumed
    Expired --> Deleted: Cleanup job runs
    Deleted --> [*]
```

```
    note right of Active
        Valid for 5 minutes
        Can only be used once
    end note
```

Ticket Validation Flow

Online Mode (Default)

```
sequenceDiagram
    participant Attendee
    participant ScannerApp
    participant CheckInService
    participant Database
    participant Cache

    Attendee->>ScannerApp: Present QR Code
    ScannerApp->>ScannerApp: Scan QR Code<br/>Extract JWT

    ScannerApp->>CheckInService: Validate Ticket<br/>POST /api/tickets/validate<br/>{jwt, scannerId}

    CheckInService->>CheckInService: Verify JWT Signature<br/>using Public Key

    alt Signature Invalid
        CheckInService-->>ScannerApp: Error: Invalid Ticket (Forged)
        ScannerApp-->>Attendee: □ ENTRY DENIED<br/>Invalid Ticket
    else Signature Valid
        CheckInService->>CheckInService: Check JWT Expiration

        alt Ticket Expired
            CheckInService-->>ScannerApp: Error: Ticket Expired
            ScannerApp-->>Attendee: □ ENTRY DENIED<br/>Ticket Expired
        else Ticket Valid
            CheckInService->>Cache: Check if already scanned<br/>(Redis: ticketId)
```

```
            alt Already Scanned
                Cache-->>CheckInService: Ticket found in scanned set
                CheckInService-->>ScannerApp: Error: Already Scanned
                ScannerApp-->>Attendee: □ ENTRY DENIED<br/>Ticket Already Used
            else First Scan
                CheckInService->>Cache: Add to scanned set<br/>SET scanned:ticketId
                CheckInService->>Database: Record Scan<br/>{ticketId, scannerId, timestamp}
                CheckInService-->>ScannerApp: Success: Entry Granted
                ScannerApp-->>Attendee: □ ENTRY GRANTED<br/>Welcome!
            end
        end
    end
```

Offline Mode (Emergency)

```
sequenceDiagram
    participant Attendee
    participant ScannerApp
    participant LocalDB
    participant PublicKey

    Note over ScannerApp: Scanner is OFFLINE<br/>No internet connection

    Attendee->>ScannerApp: Present QR Code
    ScannerApp->>ScannerApp: Scan QR Code<br/>Extract JWT

    ScannerApp->>ScannerApp: Parse JWT<br/>(header.payload.signature)

    ScannerApp->>PublicKey: Verify Signature<br/>using stored Public Key

    alt Signature Invalid
        ScannerApp-->>Attendee: □ ENTRY DENIED<br/>Invalid Ticket
    else Signature Valid
        ScannerApp->>ScannerApp: Check JWT Expiration<br/>(from exp claim)

        alt Ticket Expired
            ScannerApp-->>Attendee: □ ENTRY DENIED<br/>Ticket Expired
        else Ticket Valid
            ScannerApp->>LocalDB: Check local scanned list<br/>SELECT WHERE ticketId = ?
```

```
            alt Already Scanned Locally
                LocalDB-->>ScannerApp: Ticket found
                ScannerApp-->>Attendee: □ ENTRY DENIED<br/>Already Scanned (Local)
            else First Scan Locally
                ScannerApp->>LocalDB: Add to scanned list<br/>INSERT scan record
                ScannerApp->>LocalDB: Add to sync queue<br/>{ticketId, timestamp,
offline:true}
                ScannerApp-->>Attendee: □ ENTRY GRANTED<br/>(OFFLINE MODE)

                Note over ScannerApp: Scan queued for sync<br/>when connection returns
            end
        end
    end
```

Validation Details

JWT Signature Verification (Cryptographic Process)

```
Server Side (Ticket Generation):
1. Create payload: {ticketId, eventId, ...}
2. Sign: SIGNATURE = SHA256withRSA(header.payload, PRIVATE_KEY)
3. Result: JWT = header.payload.signature

Scanner Side (Validation):
1. Split JWT: parts = jwt.split(".")
2. Extract: header = parts[0], payload = parts[1], signature = parts[2]
3. Verify: SHA256withRSA.verify(header.payload, signature, PUBLIC_KEY)
4. If verification succeeds → Ticket is authentic
5. If verification fails → Ticket is forged/tampered
```

Online Validation Steps

```
1. Authentication Check
   - Verify scanner credentials (JWT)
   - Check if scanner is ACTIVE (not revoked)

2. Ticket Signature Verification
   - Parse JWT
   - Verify RSA signature
```

```
   - If invalid → REJECT (forged ticket)

3. Expiration Check
   - Extract exp claim from JWT
   - Compare with current time
   - If expired → REJECT

4. Duplicate Check (Redis)
   - Key: "scanned:{ticketId}"
   - Check if key exists
   - If exists → REJECT (already scanned)
   - If not exists → SET key with TTL (24 hours)

5. Database Logging
   - Insert into ticket_scans table:
     {
       ticket_id: from JWT,
       scanner_id: from request,
       scanned_at: current timestamp,
       validation_mode: "ONLINE",
       scan_result: "SUCCESS"
     }

6. Response
   - Return success with attendee details
   - Scanner displays: "Welcome, {attendee_name}!"
```

Offline Validation Steps

```
1. JWT Parsing
   - Split JWT into parts
   - Decode Base64URL payload

2. Signature Verification (Local)
   - Use stored PUBLIC_KEY
   - Verify signature cryptographically
   - If invalid → REJECT

3. Expiration Check (Local)
   - Extract exp from payload
```

```
   - Compare with device time
   - If expired → REJECT

4. Local Duplicate Check
   - Query local SQLite:
     SELECT * FROM scanned_tickets WHERE ticket_id = ?
   - If found → REJECT
   - If not found → Continue

5. Local Recording
   - INSERT INTO scanned_tickets (ticket_id, scanned_at)
   - INSERT INTO sync_queue (ticket_id, scanned_at, synced: false)

6. Response
   - Display: "Entry Granted (Offline Mode)"
   - Show sync pending indicator
```

Offline Mode Architecture

Why Offline Mode?

Events often happen in locations with poor or no internet connectivity:

- Rural areas
- Basements/underground venues
- High-attendance events (network congestion)
- Outdoor festivals
- Emergency situations

The system must continue functioning even without internet.

Offline Capabilities

```
graph TB
    subgraph "Online Operations"
        O1[Real-time duplicate detection across all gates]
        O2[Immediate sync to central database]
        O3[Live analytics dashboard]
```

```
        O4[Scanner settings updates]
    end

    subgraph "Offline Operations"
        F1[JWT signature verification]
        F2[Local duplicate detection at same gate]
        F3[Scan logging to local database]
        F4[Queue for later sync]
    end

    subgraph "Limitations in Offline Mode"
        L1[Cannot detect duplicates at other gates]
        L2[Cannot receive scanner revocations immediately]
        L3[Cannot update settings in real-time]
        L4[Relies on device clock for expiry check]
    end

    style F1 fill:#90EE90
    style F2 fill:#90EE90
    style L1 fill:#FFB6C1
    style L2 fill:#FFB6C1
```

Offline Data Flow

```
sequenceDiagram
    participant Gate1 as Scanner (Gate 1)
    participant Gate2 as Scanner (Gate 2)
    participant Server as Check-in Service

    Note over Gate1,Gate2: Both scanners go OFFLINE

    rect rgb(255, 200, 200)
        Note over Gate1,Gate2: OFFLINE PERIOD

        Gate1->>Gate1: Scan Ticket ABC<br/> Valid (first scan at Gate 1)
        Gate2->>Gate2: Scan Ticket ABC<br/> Valid (Gate 2 doesn't know)

        Note over Gate1,Gate2: PROBLEM: Same ticket scanned twice!<br/>Offline mode cannot prevent this.
```

```
    end

    Note over Gate1,Gate2: Connection restored

    rect rgb(200, 255, 200)
        Note over Gate1,Gate2: SYNC PERIOD

        Gate1->>Server: Sync scans<br/>[{ticketABC, 10:00am}]
        Server->>Server: Record: Gate 1 scanned ABC at 10:00

        Gate2->>Server: Sync scans<br/>[{ticketABC, 10:05am}]
        Server->>Server: Detect: ABC already scanned!<br/>Flag as duplicate
        Server->>Server: Create alert for investigation
    end
```

Sync Strategy

When to Sync

```
1. Automatic Sync Triggers:
   - Every N minutes (configurable, default: 15 minutes)
   - When connection restored after being offline
   - When scanner app comes to foreground
   - Before device goes to sleep

2. Manual Sync:
   - Admin can trigger sync from scanner UI
   - Force sync button available

3. Smart Sync:
   - Only sync if there are pending scans
   - Batch multiple scans in single request
   - Retry failed syncs with exponential backoff
```

Sync Process

```
sequenceDiagram
    participant Scanner
    participant LocalDB
    participant CheckInService
```

```
    participant Database

    Scanner->>LocalDB: Get pending scans<br/>SELECT * FROM sync_queue<br/>WHERE synced = false
    LocalDB-->>Scanner: Return scan records

    Scanner->>CheckInService: POST /api/scanners/sync<br/>{scannerId, scans: [...]}

    CheckInService->>CheckInService: Authenticate Scanner

    loop For each scan
        CheckInService->>Database: Check if ticket already scanned

        alt First scan of this ticket
            CheckInService->>Database: Record scan
            CheckInService->>CheckInService: Mark as SUCCESS
        else Duplicate scan
            CheckInService->>Database: Record as DUPLICATE_SCAN
            CheckInService->>CheckInService: Mark as DUPLICATE<br/>Create alert
        end
    end

    CheckInService->>CheckInService: Get latest scanner settings
    CheckInService-->>Scanner: Return {syncResults, settings}

    Scanner->>LocalDB: Update sync_queue<br/>SET synced = true
    Scanner->>Scanner: Apply new settings if changed
```

Sync Payload Example

```
POST /api/scanners/sync
Request:
{
  "scannerId": "scanner-uuid",
  "scans": [
    {
      "ticketId": "ticket-123",
      "scannedAt": "2025-11-29T10:05:00Z",
      "validationMode": "OFFLINE",
      "deviceTime": "2025-11-29T10:05:00Z"
    },
```

```
    {
      "ticketId": "ticket-456",
      "scannedAt": "2025-11-29T10:10:00Z",
      "validationMode": "OFFLINE",
      "deviceTime": "2025-11-29T10:10:00Z"
    }
  ],
  "lastSyncAt": "2025-11-29T09:00:00Z"
}

Response:
{
  "syncResults": [
    {
      "ticketId": "ticket-123",
      "status": "SUCCESS",
      "message": "Scan recorded"
    },
    {
      "ticketId": "ticket-456",
      "status": "DUPLICATE",
      "message": "Ticket already scanned at Gate B",
      "originalScanTime": "2025-11-29T10:08:00Z",
      "originalScanner": "Gate B Scanner"
    }
  ],
  "settings": {
    "offlineModeEnabled": true,
    "syncIntervalMinutes": 15,
    "maxOfflineHours": 24
  },
  "serverTime": "2025-11-29T10:30:00Z"
}
```

Conflict Resolution

```
Scenario: Same ticket scanned at multiple gates while offline


Gate A (10:05am): Scans ticket ABC - Valid 
```

```
Gate B (10:08am): Scans ticket ABC - Valid □ (doesn't know about Gate A)

Both sync at 10:30am:

Server Resolution:
1. Receive scan from Gate A (ABC at 10:05)
   - First scan seen by server
   - Record as VALID

2. Receive scan from Gate B (ABC at 10:08)
   - Server already has ABC scanned at 10:05
   - Record as DUPLICATE
   - Create alert for investigation

3. Admin Investigation:
   - Review: Was this intentional fraud?
   - Or: Genuine user scanned at wrong gate first?
   - Action: Take appropriate measures

4. Prevention for Future:
   - Reduce offline periods
   - More frequent syncs
   - Better gate coordination
```

Security Model

Cryptographic Architecture

```
graph TB
    subgraph "Server (Trust Anchor)"
        PrivateKey[RSA Private Key<br/>4096-bit<br/>NEVER leaves server]
        PublicKey[RSA Public Key<br/>Distributed to scanners]
    end

    subgraph "Ticket Generation"
        Ticket[Ticket Data<br/>JSON Payload]
        JWT[Signed JWT Token]
```

```
        QR[QR Code<br/>Contains JWT]
    end

    subgraph "Scanner Device"
        StoredPubKey[Stored Public Key]
        Verification[Signature Verification]
    end

    PrivateKey -->|Signs| JWT
    PublicKey -->|Copied to| StoredPubKey
    Ticket -->|Payload| JWT
    JWT -->|Encoded in| QR

    QR -->|Scanned| Verification
    StoredPubKey -->|Verifies| Verification

    style PrivateKey fill:#ff6666
    style PublicKey fill:#66ff66
    style Verification fill:#6666ff
```

Security Layers

Layer 1: Scanner Authentication

```
Every scanner request must include:
- Scanner credentials (JWT)
- Signed with server's private key during registration
- Contains: scannerId, scannerName, expiry (1 year)

Server validates:
1. JWT signature is valid
2. JWT not expired
3. Scanner status is ACTIVE (not revoked)
4. Scanner ID exists in database
```

Layer 2: Ticket Cryptography

```
Ticket Security Guarantees:

1. Cannot Forge Tickets
```

```
   - Requires server's private key to sign
   - Private key never leaves server
   - Scanners only have public key (can verify, not sign)

2. Cannot Tamper with Tickets
   - Any modification invalidates signature
   - Changing even 1 character breaks verification
   - Scanner detects tampering immediately

3. Cannot Reuse Expired Tickets
   - Expiry timestamp in JWT payload
   - Verified during each scan
   - Cannot be modified (signature protection)

4. Cannot Clone Tickets (Online Mode)
   - Each ticketId tracked in Redis
   - Duplicate detection across all gates
   - Scan recorded in database
```

Layer 3: Network Security

```
All API Communication:
- HTTPS/TLS 1.3 only
- Certificate pinning in mobile app
- API rate limiting
- Request signing for sensitive operations
```

Layer 4: Scanner Revocation

```
Immediate Revocation:
1. Admin marks scanner as REVOKED
2. Scanner added to revocation list (Redis)
3. Next API request from scanner → DENIED
4. Settings push: {status: "REVOKED"}
5. Scanner clears local data

Scanner receives revocation on:
- Next sync attempt
- Real-time push (if WebSocket connected)
- Settings update request
```

Attack Scenarios and Mitigations

Attack 1: Ticket Forgery

```
Attack: Attacker creates fake ticket JWT
Mitigation:
- Cannot sign without private key
- Signature verification fails
- Scanner rejects ticket
Result: □ Attack prevented
```

Attack 2: Ticket Cloning

```
Attack: User shares same ticket QR with friend

Online Mode:
- First scan: □ Valid
- Second scan: □ Duplicate detected
Result: □ Attack prevented

Offline Mode:
- Different gates: Both scans succeed locally
- Server detects on sync
- Alert generated for investigation
Result: ⚠□ Detected post-facto
```

Attack 3: Scanner Credential Theft

```
Attack: Attacker steals scanner credentials from device

Mitigation:
1. Admin revokes stolen scanner
2. Scanner credential becomes invalid
3. Server denies all requests
4. New scanner issued with new credentials

Best Practice:
- Secure credential storage (Android Keystore)
- Device encryption
- Remote wipe capability
```

```
Result: □ Contained quickly
```

Attack 4: Replay Attack

```
Attack: Attacker captures network traffic, replays requests

Mitigation:
- HTTPS prevents traffic capture
- Request timestamps checked
- Nonce validation for sensitive operations
- Short-lived sessions
Result: □ Attack prevented
```

Data Models

Database Entities

1. registration_tokens

```
CREATE TABLE registration_tokens (
    id UUID PRIMARY KEY,
    token VARCHAR(100) UNIQUE NOT NULL,
    expires_at TIMESTAMP NOT NULL,
    used BOOLEAN DEFAULT FALSE,
    used_at TIMESTAMP,
    created_by VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    validity_minutes INTEGER NOT NULL,
    scanner_name VARCHAR(200),
    notes TEXT
);

-- Indexes
CREATE INDEX idx_token ON registration_tokens(token);
CREATE INDEX idx_valid_tokens ON registration_tokens(used, expires_at)
    WHERE used = FALSE;
```

Purpose: Store time-limited tokens for scanner registration

Lifecycle:

1. Created when admin generates QR code
2. Marked as used when scanner registers
3. Cleaned up after 7 days (scheduled job)

2. scanners

```
CREATE TABLE scanners (
    id UUID PRIMARY KEY,
    scanner_id VARCHAR(100) UNIQUE NOT NULL,
    name VARCHAR(200) NOT NULL,
    credentials TEXT NOT NULL,
    status VARCHAR(20) NOT NULL, -- ACTIVE, REVOKED
    settings JSONB NOT NULL,
    device_info JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_synced_at TIMESTAMP,
    created_by VARCHAR(100)
);

-- Indexes
CREATE INDEX idx_scanner_id ON scanners(scanner_id);
CREATE INDEX idx_scanner_status ON scanners(status);
CREATE INDEX idx_last_synced ON scanners(last_synced_at);
```

Settings JSONB Structure:

```
{
  "offlineModeEnabled": false,
  "syncIntervalMinutes": 15,
  "offlineDataSource": "AUTO_SYNC",
  "maxOfflineHours": 24,
  "allowedEventIds": ["event-1", "event-2"]
}
```

3. tickets

```
CREATE TABLE tickets (
    id UUID PRIMARY KEY,
    ticket_id VARCHAR(100) UNIQUE NOT NULL,
```

```
    booking_id UUID NOT NULL,
    event_id UUID NOT NULL,
    attendee_name VARCHAR(200) NOT NULL,
    attendee_email VARCHAR(200),
    ticket_type VARCHAR(50),
    jwt_token TEXT NOT NULL,
    qr_code_base64 TEXT,
    status VARCHAR(20) NOT NULL, -- ACTIVE, SCANNED, CANCELLED
    valid_from TIMESTAMP NOT NULL,
    valid_until TIMESTAMP NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (booking_id) REFERENCES bookings(id),
    FOREIGN KEY (event_id) REFERENCES events(id)
);

-- Indexes
CREATE INDEX idx_ticket_id ON tickets(ticket_id);
CREATE INDEX idx_booking_id ON tickets(booking_id);
CREATE INDEX idx_event_id ON tickets(event_id);
CREATE INDEX idx_ticket_status ON tickets(status);
```

4. ticket_scans

```
CREATE TABLE ticket_scans (
    id UUID PRIMARY KEY,
    ticket_id VARCHAR(100) NOT NULL,
    scanner_id UUID NOT NULL,
    scanned_at TIMESTAMP NOT NULL,
    validation_mode VARCHAR(20) NOT NULL, -- ONLINE, OFFLINE
    scan_result VARCHAR(20) NOT NULL, -- SUCCESS, DUPLICATE, EXPIRED, INVALID
    device_time TIMESTAMP,
    synced_at TIMESTAMP,
    metadata JSONB,

    FOREIGN KEY (scanner_id) REFERENCES scanners(id)
);

-- Indexes
CREATE INDEX idx_ticket_scans_ticket ON ticket_scans(ticket_id);
```

```
CREATE INDEX idx_ticket_scans_scanner ON ticket_scans(scanner_id);
CREATE INDEX idx_ticket_scans_time ON ticket_scans(scanned_at);
CREATE INDEX idx_ticket_scans_result ON ticket_scans(scan_result);
```

Metadata JSONB Example:

```
{
  "attendeeName": "John Doe",
  "eventName": "Tech Conference 2025",
  "gateLocation": "Main Entrance",
  "duplicateOf": "scan-uuid-123",
  "alertGenerated": true
}
```

5. rsa_keys

```
CREATE TABLE rsa_keys (
    id UUID PRIMARY KEY,
    key_version INTEGER UNIQUE NOT NULL,
    private_key TEXT NOT NULL, -- Encrypted
    public_key TEXT NOT NULL,
    algorithm VARCHAR(20) DEFAULT 'RS256',
    key_size INTEGER DEFAULT 4096,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    rotated_at TIMESTAMP,
    status VARCHAR(20) NOT NULL -- ACTIVE, ROTATED, REVOKED
);

-- Only one active key at a time
CREATE UNIQUE INDEX idx_active_key ON rsa_keys(status)
    WHERE status = 'ACTIVE';
```

Scanner Local Database (SQLite)

```
-- Scanner credentials and config
CREATE TABLE scanner_config (
    key VARCHAR(50) PRIMARY KEY,
    value TEXT NOT NULL
);
```

```
-- Locally scanned tickets
CREATE TABLE scanned_tickets (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    ticket_id VARCHAR(100) UNIQUE NOT NULL,
    scanned_at TIMESTAMP NOT NULL,
    attendee_name VARCHAR(200),
    event_name VARCHAR(200),
    validation_result VARCHAR(20) NOT NULL
);

-- Pending sync queue
CREATE TABLE sync_queue (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    ticket_id VARCHAR(100) NOT NULL,
    scanned_at TIMESTAMP NOT NULL,
    device_time TIMESTAMP NOT NULL,
    validation_mode VARCHAR(20) NOT NULL,
    synced BOOLEAN DEFAULT FALSE,
    sync_attempts INTEGER DEFAULT 0,
    last_sync_attempt TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Indexes
CREATE INDEX idx_sync_pending ON sync_queue(synced) WHERE synced = FALSE;
CREATE INDEX idx_scanned_ticket_id ON scanned_tickets(ticket_id);
```

API Specifications

Base URL

```
Production: https://api.nexgate.com/v1
Staging: https://staging-api.nexgate.com/v1
```

Authentication

```
All requests require authentication via JWT in header:
Authorization: Bearer {scanner_credentials_jwt}
```

API Endpoints

1. Generate Registration Token

```
POST /api/registration-tokens/generate
Authorization: Bearer {admin_jwt}

Request Body:
{
  "validityMinutes": 5,
  "notes": "Gate A scanner for Main Entrance"
}

Response: 201 Created
{
  "token": "abc-123-xyz-789",
  "qrCodeBase64": "data:image/png;base64,iVBORw0KG...",
  "expiresAt": "2025-11-29T10:15:00Z",
  "validityMinutes": 5
}

Errors:
400 Bad Request - Invalid validity minutes
401 Unauthorized - Invalid admin credentials
```

2. Register Scanner

```
POST /api/scanners/register

Request Body:
{
  "token": "abc-123-xyz-789",
  "deviceName": "Gate A Scanner",
  "deviceInfo": {
    "model": "Samsung Galaxy S21",
    "osVersion": "Android 14",
```

```
    "appVersion": "1.0.0"
  }
}

Response: 201 Created
{
  "scannerId": "scanner-uuid",
  "credentials": "eyJhbGc...scanner_jwt",
  "publicKey": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCg...",
  "settings": {
    "offlineModeEnabled": false,
    "syncIntervalMinutes": 15,
    "maxOfflineHours": 24
  }
}

Errors:
400 Bad Request - Invalid or expired token
409 Conflict - Token already used
```

3. Validate Ticket (Online)

```
POST /api/tickets/validate
Authorization: Bearer {scanner_credentials}

Request Body:
{
  "jwt": "eyJhbGc...ticket_jwt",
  "scannerId": "scanner-uuid",
  "deviceTime": "2025-11-29T10:30:00Z"
}

Response: 200 OK
{
  "valid": true,
  "ticketId": "ticket-123",
  "attendeeName": "John Doe",
  "eventName": "Tech Conference 2025",
  "ticketType": "VIP",
  "message": "Entry granted"
```

```
}

Response: 400 Bad Request (Duplicate)
{
  "valid": false,
  "ticketId": "ticket-123",
  "reason": "DUPLICATE",
  "message": "Ticket already scanned",
  "originalScanTime": "2025-11-29T10:25:00Z",
  "originalScanner": "Gate B Scanner"
}

Response: 400 Bad Request (Expired)
{
  "valid": false,
  "ticketId": "ticket-123",
  "reason": "EXPIRED",
  "message": "Ticket has expired",
  "expiredAt": "2025-11-29T10:00:00Z"
}

Response: 400 Bad Request (Invalid)
{
  "valid": false,
  "reason": "INVALID_SIGNATURE",
  "message": "Ticket signature is invalid (possible forgery)"
}

Errors:
401 Unauthorized - Invalid scanner credentials
403 Forbidden - Scanner revoked
```

4. Sync Scanner

```
POST /api/scanners/sync
Authorization: Bearer {scanner_credentials}

Request Body:
{
  "scannerId": "scanner-uuid",
```

```
  "scans": [
    {
      "ticketId": "ticket-123",
      "scannedAt": "2025-11-29T10:05:00Z",
      "validationMode": "OFFLINE",
      "deviceTime": "2025-11-29T10:05:00Z"
    }
  ],
  "lastSyncAt": "2025-11-29T09:00:00Z"
}

Response: 200 OK
{
  "syncResults": [
    {
      "ticketId": "ticket-123",
      "status": "SUCCESS",
      "message": "Scan recorded"
    }
  ],
  "settings": {
    "offlineModeEnabled": true,
    "syncIntervalMinutes": 15
  },
  "serverTime": "2025-11-29T10:30:00Z",
  "pendingUpdates": []
}

Errors:
401 Unauthorized - Invalid scanner credentials
403 Forbidden - Scanner revoked
```

5. Get Scanner Settings

```
GET /api/scanners/{scannerId}/settings
Authorization: Bearer {scanner_credentials}

Response: 200 OK
{
  "offlineModeEnabled": false,
```

```
  "syncIntervalMinutes": 15,
  "offlineDataSource": "AUTO_SYNC",
  "maxOfflineHours": 24,
  "allowedEventIds": ["event-1", "event-2"]
}

Errors:
401 Unauthorized - Invalid scanner credentials
404 Not Found - Scanner not found
```

6. Revoke Scanner

```
POST /api/scanners/{scannerId}/revoke
Authorization: Bearer {admin_jwt}

Response: 200 OK
{
  "scannerId": "scanner-uuid",
  "status": "REVOKED",
  "revokedAt": "2025-11-29T10:30:00Z"
}

Errors:
401 Unauthorized - Invalid admin credentials
404 Not Found - Scanner not found
```

Database Schema

Full Schema Diagram

```
erDiagram
    REGISTRATION_TOKENS ||--o| SCANNERS : "used_by"
    SCANNERS ||--o{ TICKET_SCANS : "performs"
    TICKETS ||--o{ TICKET_SCANS : "scanned"
    EVENTS ||--o{ TICKETS : "contains"
    BOOKINGS ||--|| TICKETS : "generates"
    RSA_KEYS ||--o{ TICKETS : "signs"
```

```
REGISTRATION_TOKENS {
    uuid id PK
    varchar token UK
    timestamp expires_at
    boolean used
    timestamp used_at
    varchar created_by
    int validity_minutes
    varchar scanner_name
}

SCANNERS {
    uuid id PK
    varchar scanner_id UK
    varchar name
    text credentials
    varchar status
    jsonb settings
    jsonb device_info
    timestamp last_synced_at
}

TICKETS {
    uuid id PK
    varchar ticket_id UK
    uuid booking_id FK
    uuid event_id FK
    varchar attendee_name
    text jwt_token
    text qr_code_base64
    varchar status
    timestamp valid_from
    timestamp valid_until
}

TICKET_SCANS {
    uuid id PK
    varchar ticket_id FK
```

```
    uuid scanner_id FK
    timestamp scanned_at
    varchar validation_mode
    varchar scan_result
    jsonb metadata
}

EVENTS {
    uuid id PK
    varchar name
    timestamp event_date
    varchar venue
}

BOOKINGS {
    uuid id PK
    uuid event_id FK
    varchar customer_email
    timestamp booking_date
}

RSA_KEYS {
    uuid id PK
    int key_version UK
    text private_key
    text public_key
    varchar status
}
```

Key Relationships

```
1. registration_tokens → scanners
   - One token can register one scanner
   - Token is marked as used when scanner created

2. scanners → ticket_scans
   - One scanner performs many scans
   - Track which scanner scanned which ticket
```

```
3. tickets → ticket_scans
   - One ticket can be scanned multiple times (duplicates logged)
   - Each scan recorded separately

4. events → tickets
   - One event has many tickets
   - Tickets belong to specific event

5. bookings → tickets
   - One booking generates one or more tickets
   - Ticket inherits customer info from booking

6. rsa_keys → tickets
   - Active RSA key used to sign all new tickets
   - Key rotation supported for security
```

Deployment Architecture

Infrastructure Overview

```
graph TB
    subgraph "Client Layer"
        ScannerApp[Scanner Android App]
        AdminWeb[Admin Web Dashboard]
        CustomerApp[Customer Mobile App]
    end

    subgraph "Load Balancer"
        LB[AWS Application Load Balancer]
    end

    subgraph "Application Layer"
        API1[Check-in Service Instance 1]
        API2[Check-in Service Instance 2]
        API3[Check-in Service Instance 3]
    end
```

```
    subgraph "Cache Layer"
        Redis[(Redis Cluster<br/>Scan Deduplication)]
    end

    subgraph "Database Layer"
        PG_Primary[(PostgreSQL Primary)]
        PG_Replica[(PostgreSQL Replica)]
    end

    subgraph "Storage Layer"
        S3[S3 Bucket<br/>QR Code Images]
    end

    subgraph "Monitoring"
        CloudWatch[CloudWatch Logs]
        Grafana[Grafana Dashboard]
    end

    ScannerApp --> LB
    AdminWeb --> LB
    CustomerApp --> LB

    LB --> API1
    LB --> API2
    LB --> API3

    API1 --> Redis
    API2 --> Redis
    API3 --> Redis

    API1 --> PG_Primary
    API2 --> PG_Primary
    API3 --> PG_Primary

    PG_Primary --> PG_Replica

    API1 --> S3
    API2 --> S3
    API3 --> S3
```

```
    API1 --> CloudWatch
    API2 --> CloudWatch
    API3 --> CloudWatch

    CloudWatch --> Grafana
```

Deployment Configuration

Production Environment

```
Check-in Service:
  Instances: 3 (Auto-scaling: 2-10)
  Instance Type: t3.large (2 vCPU, 8 GB RAM)
  Deployment: Blue-Green with ECS
  Health Check: /api/health every 30s

Database:
  Type: Amazon RDS PostgreSQL 15
  Instance: db.r6g.xlarge (4 vCPU, 32 GB RAM)
  Storage: 500 GB SSD (auto-scaling enabled)
  Replication: 1 read replica in different AZ
  Backup: Daily automated backups, 30-day retention

Redis:
  Type: Amazon ElastiCache
  Node Type: cache.r6g.large (2 vCPU, 13 GB RAM)
  Cluster: 3 nodes (1 primary, 2 replicas)
  Persistence: AOF enabled

Load Balancer:
  Type: Application Load Balancer
  SSL: AWS Certificate Manager
  Zones: Multi-AZ deployment

Monitoring:
  CloudWatch: All application and infrastructure metrics
  Grafana: Custom dashboards for scan analytics
  PagerDuty: Alert escalation
```

```
Backups:
  Database: Daily automated + on-demand
  Redis: Daily snapshots
  S3: Versioning enabled
```

Scaling Strategy

```
Horizontal Scaling (API Instances):
- Metric: CPU > 70% or Request Count > 1000/min
- Scale up: Add 1 instance
- Scale down: Remove 1 instance if CPU < 30%
- Min instances: 2
- Max instances: 10

Database Scaling:
- Vertical: Upgrade instance type during low-traffic window
- Horizontal: Add read replicas for reporting/analytics
- Connection pooling: HikariCP with max 100 connections

Redis Scaling:
- Vertical: Upgrade node type
- Horizontal: Add replica nodes
- Cluster mode: Enable for > 10M keys

Regional Expansion:
- Deploy Check-in Service in multiple AWS regions
- Use Route 53 for geo-routing
- Replicate database across regions (read replicas)
```

Disaster Recovery

```
RTO (Recovery Time Objective): 1 hour
RPO (Recovery Point Objective): 5 minutes

Recovery Procedures:

1. Database Failure:
   - Automatic failover to replica (< 2 minutes)
```

```
   - Promote replica to primary
   - Update application config
   - Restore read replica from backup

2. API Service Failure:
   - Auto Scaling Group spawns new instances
   - Load Balancer routes to healthy instances
   - Failed instances terminated and replaced

3. Redis Failure:
   - Automatic failover to replica
   - Application continues with slight latency
   - Rebuild cache from database if needed

4. Complete Region Failure:
   - Route 53 failover to backup region
   - Promote backup region database to primary
   - Update scanner apps via backend config
```

Monitoring and Analytics

Key Metrics

```
Scanner Metrics:
- Total active scanners
- Scanners online vs offline
- Scan rate per scanner
- Sync frequency and success rate
- Average offline duration

Ticket Metrics:
- Total scans per event
- Valid scans vs duplicates vs invalid
- Scan success rate
- Average scan time (online vs offline)
- Peak scan throughput
```

```
Performance Metrics:
- API response time (p50, p95, p99)
- Database query performance
- Redis hit/miss ratio
- Error rate by endpoint

Security Metrics:
- Invalid ticket attempts
- Scanner authentication failures
- Duplicate scan alerts
- Suspicious patterns (multiple duplicates)
```

Alerting

```
Critical Alerts (PagerDuty):
- Database connection pool exhausted
- API error rate > 5%
- Redis cluster down
- Duplicate scan rate > 10%

Warning Alerts (Slack):
- Scanner offline > 30 minutes
- Sync failure rate > 20%
- API response time > 2s (p95)
- Database replication lag > 10s

Info Alerts (Email):
- Daily scan summary
- Weekly duplicate report
- Monthly scanner registration report
```

Appendix

Glossary

RSA (Rivest-Shamir-Adleman): Asymmetric encryption algorithm used for digital signatures

JWT (JSON Web Token): Compact, URL-safe means of representing claims between two parties

QR Code: Two-dimensional barcode that can be scanned by cameras

Scanner Registration: Process of linking a scanner device to the system

Offline Mode: Scanner operation without internet connectivity

Duplicate Scan: Attempt to scan the same ticket multiple times

Sync Queue: Local storage of scans waiting to be sent to server

Public Key: Cryptographic key used to verify signatures (safe to distribute)

Private Key: Cryptographic key used to create signatures (must remain secret)

References

- JWT RFC: https://datatracker.ietf.org/doc/html/rfc7519
- RSA Cryptography: PKCS #1 v2.2
- QR Code Standard: ISO/IEC 18004:2015
- Android Keystore: https://developer.android.com/training/articles/keystore

Change Log

Version 1.0 (2025-11-29)

- Initial architecture document
- Complete system design
- All core flows documented

Document Maintained By: Nexgate Platform Team
Last Updated: November 29, 2025
Next Review: December 29, 2025

Revision #1
Created 29 November 2025 07:26:23 by Admin Qbit
Updated 11 December 2025 09:32:31 by Admin Qbit