

Bus Tracker System

System Design Document

A Comprehensive School Bus Tracking and
Student Attendance Management System

Version: 1.0

Date: November 20, 2025

Technology Stack:

FastAPI — React — MongoDB — Raspberry Pi

Document Purpose:

This document provides a comprehensive technical design of the Bus Tracker System architecture, components, data models, and integration patterns.

Contents

1	Executive Summary	4
1.1	Key Features	4
1.2	System Goals	4
2	System Architecture	5
2.1	High-Level Architecture	5
2.2	Architecture Diagram	5
2.3	Technology Stack	5
3	Data Model	7
3.1	Database Schema Overview	7
3.1.1	Users Collection	7
3.1.2	Students Collection	7
3.1.3	Attendance Collection	7
3.1.4	Buses Collection	8
3.1.5	Routes Collection	8
3.1.6	Stops Collection	8
3.1.7	Notifications Collection	9
3.1.8	Holidays Collection	9
3.1.9	Device Keys Collection	9
3.2	Data Relationships	9
3.3	Database Constraints	10
4	API Architecture	11
4.1	REST API Design	11
4.2	API Endpoint Categories	11
4.2.1	Authentication Endpoints	11
4.2.2	Student Management	11
4.2.3	User Management	11
4.2.4	Attendance & Tracking	11
4.2.5	Bus & Route Management	12
4.2.6	Notifications	12
4.2.7	Holidays	12
4.2.8	Device API (IoT)	12
4.2.9	Backup System	12
4.2.10	Photo Management	13
4.3	Authentication Flow	13
4.4	Device Authentication	13
5	Frontend Architecture	14
5.1	Component Structure	14
5.2	State Management	14
5.3	Dashboard Features	14
5.3.1	Parent Dashboard	14
5.3.2	Teacher Dashboard	15
5.3.3	Admin Dashboard	15

5.4	UI/UX Design Principles	15
5.5	Map Integration	15
6	IoT Integration	17
6.1	Raspberry Pi Architecture	17
6.2	Pi Software Architecture	17
6.3	Attendance Workflow	17
6.4	Face Verification	18
6.5	Offline Operation	18
6.6	GPS Handling	18
6.7	Device Security	18
7	Security Architecture	19
7.1	Authentication & Authorization	19
7.2	Role-based Permissions	19
7.3	Data Protection	19
7.4	Backup Security	19
8	Backup System	21
8.1	Backup Strategy	21
8.2	Backup Process	21
8.3	Restore Process	21
8.4	Backup Monitoring	22
9	Photo Management	23
9.1	Photo Organization	23
9.2	Photo Upload	23
9.3	Photo Serving	23
9.4	Attendance Photos	23
10	Deployment Architecture	25
10.1	Container Architecture	25
10.2	Service Configuration	25
10.2.1	Backend Service	25
10.2.2	Frontend Service	25
10.3	Environment Variables	25
10.3.1	Backend (.env)	25
10.3.2	Frontend (.env)	26
10.4	Ingress Routing	26
10.5	Service Control	26
11	Performance Considerations	27
11.1	Backend Optimization	27
11.2	Frontend Optimization	27
11.3	Database Optimization	27
11.4	IoT Optimization	27

12 Testing Strategy	28
12.1 Testing Levels	28
12.2 Backend Testing	28
12.3 Frontend Testing	28
12.4 IoT Testing	28
13 Future Enhancements	29
13.1 Planned Features	29
13.2 Technical Improvements	29
13.3 Security Enhancements	29
14 Appendices	30
14.1 Appendix A: Database Indexes	30
14.2 Appendix B: API Status Codes	30
14.3 Appendix C: Useful Commands	30
14.4 Appendix D: Dependencies	31
14.4.1 Backend (requirements.txt)	31
14.4.2 Frontend (package.json highlights)	31
15 Glossary	32
16 Conclusion	33
16.1 Key Achievements	33
16.2 System Benefits	33
16.3 Technology Highlights	33

1 Executive Summary

The Bus Tracker System is a modern, full-stack web application designed to streamline school transportation management through real-time GPS tracking, automated RFID-based student attendance, and role-based dashboards for parents, teachers, and administrators.

1.1 Key Features

- **Real-time Bus Tracking:** Live GPS monitoring with interactive maps
- **Automated Attendance:** RFID scanning with photo capture and face verification
- **Role-based Dashboards:** Customized interfaces for Parents, Teachers, and Admins
- **Instant Notifications:** Real-time alerts for identity mismatches and updates
- **IoT Integration:** Raspberry Pi devices with RFID readers and cameras
- **Backup System:** Production-ready backups with SHA256 integrity verification

1.2 System Goals

- Enhance student safety through real-time tracking and verification
- Automate attendance recording and reduce manual errors
- Provide instant visibility to parents on their children's status
- Enable efficient school transportation management
- Support offline operation with graceful degradation

2 System Architecture

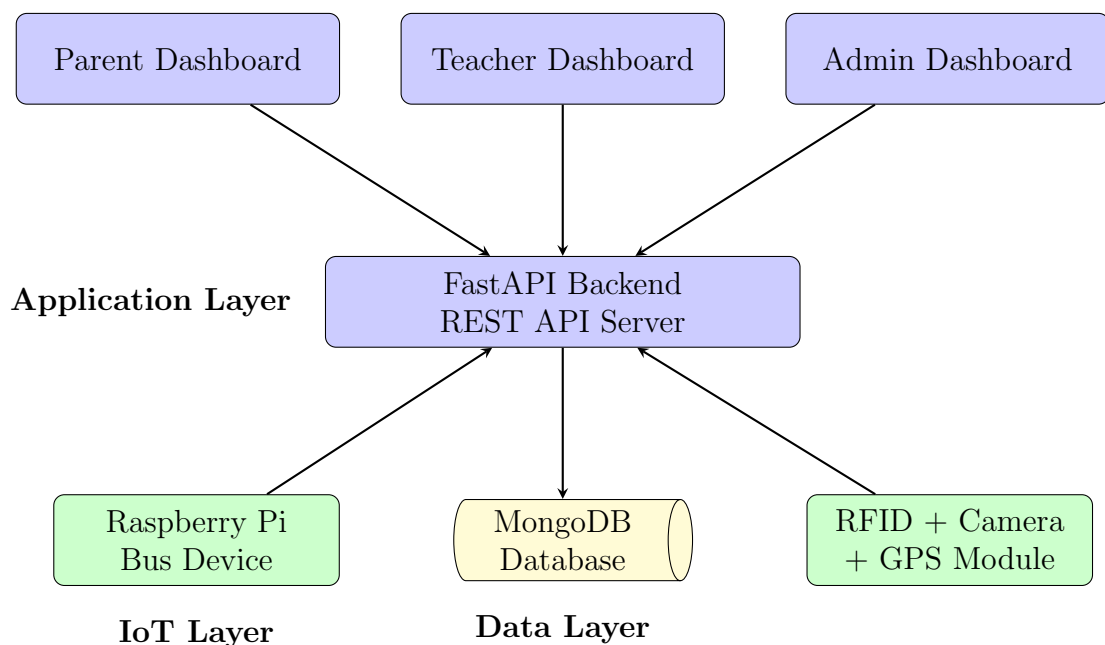
2.1 High-Level Architecture

The Bus Tracker System follows a three-tier architecture pattern with IoT integration:

1. **Presentation Layer:** React-based responsive web application
2. **Application Layer:** FastAPI REST API server
3. **Data Layer:** MongoDB database with async operations
4. **IoT Layer:** Raspberry Pi devices on buses

2.2 Architecture Diagram

Presentation Layer (React)



2.3 Technology Stack

Frontend Technologies

- **Framework:** React 18.x
- **UI Library:** Radix UI, Tailwind CSS
- **Maps:** Leaflet with OpenStreetMap
- **State Management:** React Context API
- **Build Tool:** Create React App with Craco

Backend Technologies

- **Framework:** FastAPI (Python 3.9+)
- **Database Driver:** Motor (Async MongoDB)
- **Authentication:** bcrypt, session-based auth
- **AI/ML:** DeepFace (face verification), OpenCV
- **Email:** aiosmtplib (async SMTP)

Database & Infrastructure

- **Database:** MongoDB 6.0+
- **Process Manager:** Supervisor
- **Web Server:** Uvicorn (ASGI)
- **Deployment:** Kubernetes with Ingress

IoT Technologies

- **Hardware:** Raspberry Pi 3/4
- **RFID Reader:** MFRC522 module
- **Camera:** IP WebCamera
- **GPS:** M80 Pro GPS module
- **Connectivity:** SIM800A GSM/GPRS module/Wi-fi fallback

3 Data Model

3.1 Database Schema Overview

The system uses MongoDB with the following collections:

3.1.1 Users Collection

Stores all system users (admins, teachers, parents).

```
1 {  
2   "user_id": "uuid",  
3   "email": "user@school.com",  
4   "password_hash": "bcrypt_hash",  
5   "role": "admin|teacher|parent",  
6   "name": "John Doe",  
7   "phone": "+1-555-0001",  
8   "photo_path": "backend/photos/parents/uuid.jpg",  
9   "address": "123 Main St",  
10  "student_ids": ["uuid1", "uuid2"], # for parents  
11  "assigned_class": "5",             # for teachers  
12  "assigned_section": "A",           # for teachers  
13  "is_elevated_admin": true,         # for admins  
14  "created_at": "2025-01-01T00:00:00Z"  
15 }
```

Listing 1: User Model

3.1.2 Students Collection

Stores student information with attendance tracking.

```
1 {  
2   "student_id": "uuid",  
3   "name": "Emma Johnson",  
4   "roll_number": "G5A-001",  
5   "class_name": "5",  
6   "section": "A",  
7   "parent_id": "uuid",  
8   "teacher_id": "uuid",  
9   "bus_id": "BUS-001",  
10  "stop_id": "uuid",  
11  "rfid_tag": "E200001F9A8D0F23",  
12  "photo_path": "backend/photos/students/uuid/profile.jpg",  
13  "attendance_path": "backend/photos/students/uuid/attendance/",  
14  "embedding": "base64_face_embedding",  
15  "phone": "+1-555-1001",  
16  "address": "456 Oak Ave",  
17  "created_at": "2025-01-01T00:00:00Z"  
18 }
```

Listing 2: Student Model

3.1.3 Attendance Collection

Records all attendance events with photos.

```
1 {
2   "attendance_id": "uuid",
3   "student_id": "uuid",
4   "bus_id": "BUS-001",
5   "date": "2025-01-15",
6   "time_period": "AM|PM",
7   "attendance_status": "yellow|green",
8   "scan_time": "2025-01-15T07:30:00Z",
9   "photo_url": "attendance/2025-01-15_AM.jpg",
10  "verified": true,
11  "confidence_score": 0.95,
12  "location": {
13    "latitude": 28.6139,
14    "longitude": 77.2090
15  }
16 }
```

Listing 3: Attendance Model

3.1.4 Buses Collection

Stores bus information and current location.

```
1 {
2   "bus_id": "BUS-001",
3   "bus_number": "DL-01-AB-1234",
4   "driver_name": "Michael Brown",
5   "driver_phone": "+1-555-3001",
6   "capacity": 40,
7   "route_id": "uuid",
8   "current_location": {
9     "latitude": 28.6139,
10    "longitude": 77.2090,
11    "timestamp": "2025-01-15T08:00:00Z"
12  }
13 }
```

Listing 4: Bus Model

3.1.5 Routes Collection

Defines bus routes with stops.

```
1 {
2   "route_id": "uuid",
3   "route_name": "Route A - North District",
4   "stop_ids": ["uuid1", "uuid2", "uuid3"],
5   "map_path": [[lat1, lon1], [lat2, lon2], ...]
6 }
```

Listing 5: Route Model

3.1.6 Stops Collection

Bus stop locations.

```
1 {  
2   "stop_id": "uuid",  
3   "stop_name": "Main Gate North",  
4   "latitude": 28.6139,  
5   "longitude": 77.2090,  
6   "order_index": 0  
7 }
```

Listing 6: Stop Model

3.1.7 Notifications Collection

User notifications and alerts.

```
1 {  
2   "notification_id": "uuid",  
3   "user_id": "uuid",  
4   "type": "identity_mismatch|welcome|update",  
5   "title": "Identity Verification Alert",  
6   "message": "Face verification failed for Emma",  
7   "read": false,  
8   "created_at": "2025-01-15T07:30:00Z"  
9 }
```

Listing 7: Notification Model

3.1.8 Holidays Collection

School holiday calendar.

```
1 {  
2   "holiday_id": "uuid",  
3   "name": "Christmas Day",  
4   "date": "2025-12-25",  
5   "description": "National holiday"  
6 }
```

Listing 8: Holiday Model

3.1.9 Device Keys Collection

API keys for IoT devices.

```
1 {  
2   "device_id": "uuid",  
3   "bus_id": "BUS-001",  
4   "device_name": "Pi Scanner - Bus 001",  
5   "key_hash": "bcrypt_hash",  
6   "created_at": "2025-01-01T00:00:00Z"  
7 }
```

Listing 9: DeviceKey Model

3.2 Data Relationships

- **Users (Parents) → Students:** One-to-Many (via student_ids)

- **Users (Teachers) → Students:** One-to-Many (via assigned_class/section)
- **Students → Attendance:** One-to-Many
- **Buses → Routes:** Many-to-One
- **Routes → Stops:** One-to-Many
- **Students → Stops:** Many-to-One
- **Device Keys → Buses:** One-to-One

3.3 Database Constraints

- **Unique Indexes:**
 - users.email
 - students.(class_name, section, roll_number) - Composite
 - buses.bus_number
 - device_keys.bus_id
- **Dependency Safeguards:**
 - Students cannot be deleted if attendance records exist
 - Parents cannot be deleted if students are linked
 - Teachers cannot be deleted if students are assigned
 - Buses cannot be deleted if students are assigned
 - Routes cannot be deleted if buses are using them
 - Stops cannot be deleted if students or routes reference them

4 API Architecture

4.1 REST API Design

The backend exposes a RESTful API with the following characteristics:

API Principles

- **Base URL:** /api (Kubernetes ingress routing)
- **Authentication:** Session-based with httpOnly cookies
- **Authorization:** Role-based access control (RBAC)
- **Response Format:** JSON
- **Error Handling:** HTTP status codes with descriptive messages
- **Async Operations:** Non-blocking I/O with Motor driver

4.2 API Endpoint Categories

4.2.1 Authentication Endpoints

1	POST	/api/auth/login	# User login
2	POST	/api/auth/logout	# User logout
3	GET	/api/auth/me	# Get current user info
4	POST	/api/auth/verify-code	# Email verification (optional)

4.2.2 Student Management

1	GET	/api/students	# List students (role-filtered)
2	GET	/api/students/{id}	# Get student details
3	POST	/api/students	# Create student (admin)
4	PUT	/api/students/{id}	# Update student (admin)
5	DELETE	/api/students/{id}	# Delete student (admin)
6	GET	/api/students/class-sections	# Get existing class-sections
7	PUT	/api/students/{id}/register-rfid	# Register RFID tag

4.2.3 User Management

1	GET	/api/users	# List users (admin)
2	POST	/api/users	# Create user (admin)
3	PUT	/api/users/{id}	# Update user (admin)
4	DELETE	/api/users/{id}	# Delete user (admin)
5	GET	/api/parents/all	# List all parents
6	GET	/api/parents/unlinked	# List parents without children

4.2.4 Attendance & Tracking

1	POST	/api/scan_event	# Record attendance scan (device)
2	POST	/api/update_location	# Update bus GPS (device)
3	GET	/api/get_attendance	# Get attendance grid
4	GET	/api/get_bus_location	# Get current bus location

4.2.5 Bus & Route Management

```

1 GET      /api/buses                # List buses
2 GET      /api/buses/{id}          # Get bus details
3 POST     /api/buses                # Create bus (admin)
4 PUT      /api/buses/{id}          # Update bus (admin)
5 DELETE   /api/buses/{id}          # Delete bus (admin)
6 GET      /api/buses/{id}/stops    # Get stops for bus route
7
8 GET      /api/routes              # List routes
9 GET      /api/routes/{id}         # Get route details
10 POST    /api/routes              # Create route (admin)
11 PUT     /api/routes/{id}         # Update route (admin)
12 DELETE  /api/routes/{id}         # Delete route (admin)
13
14 GET      /api/stops              # List stops
15 POST    /api/stops              # Create stop (admin)
16 PUT     /api/stops/{id}         # Update stop (admin)
17 DELETE  /api/stops/{id}         # Delete stop (admin)

```

4.2.6 Notifications

```

1 GET      /api/get_notifications    # List user notifications
2 PUT      /api/mark_notification_read/{id} # Mark as read
3 DELETE   /api/notifications/{id}  # Delete notification

```

4.2.7 Holidays

```

1 GET      /api/admin/holidays      # List holidays
2 POST     /api/admin/holidays      # Create holiday (admin)
3 PUT      /api/admin/holidays/{id} # Update holiday (admin)
4 DELETE   /api/admin/holidays/{id} # Delete holiday (admin)

```

4.2.8 Device API (IoT)

```

1 POST     /api/device/register      # Register device (
  admin)
2 GET      /api/device/list          # List devices (admin)
3 GET      /api/students/{id}/embedding # Get face embedding (
  device)
4 GET      /api/students/embedding-by-rfid # Get embedding by RFID
  (device)

```

4.2.9 Backup System

```

1 GET      /api/admin/backups/status # Current backup status
2 GET      /api/admin/backups/list   # List all backups
3 GET      /api/admin/backups/health # Backup system health
4 POST     /api/admin/backups/trigger # Manual backup trigger
5 POST     /api/admin/backups/verify/{id} # Verify backup integrity
6 POST     /api/admin/backups/restore/{id} # Restore from backup

```

4.2.10 Photo Management

1	PUT	/api/users/me/photo	# Upload user photo
2	PUT	/api/students/{id}/photo	# Upload student photo
3	GET	/photos/*	# Static file serving

4.3 Authentication Flow

1. User submits credentials via POST /api/auth/login
2. Backend validates credentials with bcrypt
3. If valid, creates session token (UUID)
4. Session stored in memory with user data
5. Returns session_token as httpOnly cookie
6. Subsequent requests include cookie automatically
7. Backend validates session on protected endpoints
8. Logout clears session from memory and cookie

4.4 Device Authentication

IoT devices use API key authentication:

1. Admin registers device via POST /api/device/register
2. System generates 64-character API key
3. Key is hashed with bcrypt before storage
4. Device includes X-API-Key header in requests
5. Backend validates key using verify_device_key() dependency
6. Device can only access device-specific endpoints

5 Frontend Architecture

5.1 Component Structure

The React frontend is organized into the following structure:

```

1 frontend/
2 |-- src/
3     |-- components/
4         |-- AdminDashboardNew.jsx      # Admin dashboard
5         |-- TeacherDashboardNew.jsx    # Teacher dashboard
6         |-- ParentDashboard.jsx        # Parent dashboard
7         |-- LoginPage.jsx              # Login page
8         |-- AddStudentModal.jsx        # Student creation
9         |-- EditStudentModalEnhanced.jsx # Student editing
10        |-- StudentDetailModal.jsx     # Student details
11        |-- AddHolidayModal.jsx        # Holiday creation
12        |-- EditHolidayModal.jsx       # Holiday editing
13        |-- PhotoViewerModal.jsx       # Photo viewer
14        |-- UserProfileHeader.jsx      # User profile header
15        |-- StudentCard.jsx            # Student card component
16        |-- DeleteConfirmationDialog.jsx # Delete confirmation
17        |-- HolidaysManagementModal.jsx # Holidays management
18    |-- App.js                          # Main app component
19    |-- index.js                        # Entry point
20    |-- index.css                      # Global styles
21 |-- public/
22    |-- index.html

```

Listing 10: Frontend Directory Structure

5.2 State Management

The application uses React Context API for global state:

- **UserContext:** Current user session and profile
- **Component State:** Local state with useState hook
- **API Calls:** Direct fetch calls with async/await

5.3 Dashboard Features

5.3.1 Parent Dashboard

- View children's information and photos
- Live bus tracking on interactive map
- Monthly attendance calendar with click-to-view photos
- Real-time notifications
- Student detail cards with bus and stop information

5.3.2 Teacher Dashboard

- View assigned students list
- Today's AM/PM attendance status
- Search and filter students
- View student details and attendance history
- Access to route and bus information

5.3.3 Admin Dashboard

- Complete CRUD operations for:
 - Students with class-section management
 - Users (admins, teachers, parents)
 - Buses and routes
 - Holidays
- System statistics and overview
- Backup management and monitoring
- Device registration and management
- Elevated admin permissions

5.4 UI/UX Design Principles

Design System

- **Color Scheme:** Role-specific colors (blue for parents, teal for teachers, indigo for admins)
- **Typography:** System fonts with Tailwind CSS utilities
- **Icons:** Lucide React icon library
- **Responsive:** Mobile-first design with breakpoints
- **Animations:** Smooth transitions with Tailwind CSS
- **Accessibility:** ARIA labels and keyboard navigation

5.5 Map Integration

The system uses Leaflet with OpenStreetMap for real-time bus tracking:

- Bus marker with custom icon
- Route path polyline visualization
- Stop markers with popup information
- Auto-zoom to fit route bounds

- Toggle route display on/off
- GPS fallback indicator (red marker with question mark)

6 IoT Integration

6.1 Raspberry Pi Architecture

Each bus is equipped with a Raspberry Pi device running custom Python software:

Pi Hardware Components

- **Main Controller:** Raspberry Pi 3 Model B+ or 4
- **RFID Reader:** MFRC522 (SPI interface)
- **Camera:** IP Webcam
- **GPS Module:** M80 Pro (UART)
- **Connectivity:** SIM800A GSM/GPRS module
- **Power:** 5V/3A adapter or vehicle power

6.2 Pi Software Architecture

The Raspberry Pi runs three main scripts:

1. **pi_server.py:** Main boarding scanner controller
2. **pi_hardware_auto.py:** Hardware backend (GPIO, RFID, Camera, GPS)
3. **pi_simulated.py:** Simulation backend (for testing)

6.3 Attendance Workflow

1. Student taps RFID card on reader
2. Pi reads RFID tag and looks up student in local cache
3. Camera captures student photo
4. Pi performs face verification using DeepFace
5. Confidence score calculated (threshold: 0.70)
6. GPS coordinates captured
7. Data uploaded to backend via API:
 - Student ID
 - Photo (base64 encoded)
 - Verification status
 - GPS coordinates
 - Scan type (yellow or green)
8. Backend creates attendance record
9. If face verification fails, notification sent to parent

6.4 Face Verification

The system uses DeepFace library with Facenet model:

- **Embedding Generation:** 128-dimension face embeddings
- **Similarity Metric:** Cosine similarity
- **Threshold:** 0.70 (adjustable)
- **Caching:** Embeddings stored locally in JSON file
- **Fallback:** API fetch if local embedding missing
- **Retry Logic:** 2 failures → fresh API fetch → 3 retries

6.5 Offline Operation

The Pi system supports offline operation:

- Local embedding cache for face verification
- RFID-to-student mapping stored locally
- Queue system for failed uploads (future enhancement)
- Graceful degradation when API unavailable

6.6 GPS Handling

The system handles GPS failures gracefully:

- **Valid GPS:** Full coordinates recorded
- **No GPS Fix:** Null coordinates sent, system continues
- **Frontend:** Displays red question mark indicator
- **Backend:** Accepts null latitude/longitude

6.7 Device Security

- API key authentication for all device requests
- Keys stored securely in device configuration file
- Each device mapped to specific bus (1:1 relationship)
- Device can only update location for assigned bus
- Keys cannot be retrieved after generation

7 Security Architecture

7.1 Authentication & Authorization

Security Layers

- **Password Hashing:** bcrypt with automatic salt
- **Session Management:** In-memory sessions with UUID tokens
- **Cookie Security:** httpOnly, sameSite=lax, 24h expiry
- **Role-based Access:** Three roles with distinct permissions
- **Device Authentication:** API key (64-char hex) with bcrypt hashing
- **Input Validation:** Pydantic models for all requests

7.2 Role-based Permissions

Feature	Parent	Teacher	Admin
View own children		-	
View assigned students	-		
View all students	-	-	
Create/edit students	-	-	
Delete students	-	-	
User management	-	-	
Bus/route management	-	-	
Holiday management	-	-	
Device registration	-	-	
Backup management	-	-	
View notifications			
Track bus location			

7.3 Data Protection

- **Encryption in Transit:** HTTPS required in production
- **Password Storage:** Never store plaintext passwords
- **API Response:** Password hashes excluded from responses
- **File Upload:** Type and size validation
- **Photo Access:** Static file serving with path validation
- **SQL Injection:** Protected by MongoDB (NoSQL)
- **XSS Protection:** React auto-escaping, CSP headers

7.4 Backup Security

- **Integrity Verification:** SHA256 checksums for all backups
- **Backup Encryption:** File-level encryption (future enhancement)

- **Access Control:** Admin-only backup operations
- **Audit Trail:** All backup operations logged
- **Restore Validation:** Checksum verification before restore

8 Backup System

8.1 Backup Strategy

The system implements a production-ready backup solution:

Backup Features

- **Auto-rotation:** Keeps last 10 backups, auto-deletes old ones
- **Integrity Checks:** SHA256 checksums for verification
- **Metadata Tracking:** Size, date, collection counts, status
- **Health Monitoring:** Last backup time, disk usage alerts
- **Manual Trigger:** Admin can trigger backup anytime
- **Restore Capability:** Full database restore with validation
- **Frontend Integration:** Status display in admin dashboard

8.2 Backup Process

1. Backup triggered (manual or scheduled)
2. Generate unique backup ID (timestamp-based)
3. Export all MongoDB collections to JSON
4. Calculate SHA256 checksum of backup file
5. Store metadata (size, counts, checksum)
6. Rotate old backups (keep last 10)
7. Update backup status in database

8.3 Restore Process

1. Admin selects backup to restore
2. System verifies backup integrity (checksum)
3. If valid, clears existing database collections
4. Imports data from backup file
5. Validates restore success
6. Returns collection counts

8.4 Backup Monitoring

Admins can monitor backup health via dashboard:

- Last backup date and time
- Total number of backups
- Total backup storage size
- Backup history with individual sizes
- Health status indicators
- Disk usage warnings

9 Photo Management

9.1 Photo Organization

Photos are organized by user role and type:

```

1 backend/photos/
2 |-- students/
3     |-- {student_id}/
4         |-- profile.jpg           # Profile photo
5         |-- attendance/
6             |-- 2025-01-15_AM.jpg # AM scan photo
7             |-- 2025-01-15_PM.jpg # PM scan photo
8 |-- parents/
9     |-- {user_id}.jpg             # Parent photo
10 |-- teachers/
11     |-- {user_id}.jpg             # Teacher photo
12 |-- admins/
13     |-- {user_id}.jpg             # Admin photo

```

Listing 11: Photo Directory Structure

9.2 Photo Upload

- **Allowed Formats:** JPEG, PNG
- **Size Limit:** 10 MB per file
- **Validation:** Content-type checking
- **Storage:** Local filesystem
- **Database:** photo_path field stores relative path
- **URL Conversion:** Backend converts path to accessible URL

9.3 Photo Serving

- **Static Files:** Mounted at /photos endpoint
- **Direct Access:** /photos/students/id/profile.jpg
- **Content-Type:** Automatic MIME type detection
- **Caching:** Browser caching enabled
- **Fallback:** UI shows user initials if photo missing

9.4 Attendance Photos

- Captured during RFID scan on Pi
- Uploaded as base64 encoded string
- Backend decodes and saves to attendance folder

- Naming convention: YYYY-MM-DD_AM.jpg or YYYY-MM-DD_PM.jpg
- Displayed in attendance calendar on click
- Used for face verification with DeepFace

10 Deployment Architecture

10.1 Container Architecture

The system runs in a Kubernetes cluster:

Deployment Components

- **Backend:** FastAPI on Uvicorn (port 8001)
- **Frontend:** React dev server (port 3000)
- **MongoDB:** Database service (port 27017)
- **Process Manager:** Supervisor for service control
- **Ingress:** Kubernetes ingress for routing

10.2 Service Configuration

10.2.1 Backend Service

```
1 # Supervisor configuration
2 [program:backend]
3 command=uvicorn server:app --host 0.0.0.0 --port 8001
4 directory=/app/backend
5 autostart=true
6 autorestart=true
```

10.2.2 Frontend Service

```
1 # Supervisor configuration
2 [program:frontend]
3 command=yarn start
4 directory=/app/frontend
5 autostart=true
6 autorestart=true
7 environment=PORT=3000
```

10.3 Environment Variables

10.3.1 Backend (.env)

```
1 MONGO_URL=mongodb://localhost:27017
2 DB_NAME=bus_tracker
3 BACKEND_BASE_URL=https://app.example.com
4 CORS_ORIGINS=*
5 TIMEZONE=Asia/Kolkata
6 EMAIL_AUTH_ENABLED=false
7 NEW_USER_EMAIL_ENABLED=true
8 SMTP_HOST=smtp.gmail.com
9 SMTP_PORT=587
10 SMTP_USER=
11 SMTP_PASS=
```

10.3.2 Frontend (.env)

```
1 REACT_APP_BACKEND_URL=https://app.example.com
```

10.4 Ingress Routing

Kubernetes ingress routes requests:

- `/api/*` → Backend service (port 8001)
- `/*` → Frontend service (port 3000)

10.5 Service Control

```
1 # Check service status
2 sudo supervisorctl status
3
4 # Restart services
5 sudo supervisorctl restart backend
6 sudo supervisorctl restart frontend
7 sudo supervisorctl restart all
8
9 # View logs
10 tail -f /var/log/supervisor/backend.out.log
11 tail -f /var/log/supervisor/frontend.out.log
```

11 Performance Considerations

11.1 Backend Optimization

- **Async Operations:** All database calls are async with Motor
- **Connection Pooling:** MongoDB connection reuse
- **Indexing:** Proper indexes on frequently queried fields
- **Pagination:** Large result sets paginated (future)
- **Caching:** In-memory session storage
- **Static Files:** Efficient serving with FastAPI StaticFiles

11.2 Frontend Optimization

- **Code Splitting:** Lazy loading for large components (future)
- **Image Optimization:** Responsive images with proper sizing
- **Memoization:** React.memo for expensive components
- **Debouncing:** Search inputs debounced (300ms)
- **Virtual Scrolling:** For large student lists (future)

11.3 Database Optimization

- **Indexes:**
 - users.email (unique)
 - students.(class_name, section, roll_number) (compound unique)
 - attendance.(student_id, date)
 - notifications.(user_id, created_at)
- **Data Enrichment:** Single queries with projection
- **Aggregation:** MongoDB aggregation pipeline for complex queries

11.4 IoT Optimization

- **Local Caching:** Face embeddings cached on Pi
- **Batch Operations:** Future enhancement for offline queuing
- **Image Compression:** Photos compressed before upload
- **Retry Logic:** Exponential backoff for failed requests

12 Testing Strategy

12.1 Testing Levels

1. **Unit Testing:** Individual functions and components
2. **Integration Testing:** API endpoint testing
3. **End-to-End Testing:** Full user workflows (removed test files)
4. **Manual Testing:** UI/UX and device testing

12.2 Backend Testing

- **Framework:** pytest
- **Coverage:** API endpoints, models, utilities
- **Mocking:** Database connections, external services
- **Fixtures:** Test data and setup

12.3 Frontend Testing

- **Framework:** Jest + React Testing Library
- **Coverage:** Components, hooks, utilities
- **Snapshot Testing:** UI component rendering
- **Interaction Testing:** User actions and events

12.4 IoT Testing

- **Simulation Mode:** Test without hardware (pi_simulated.py)
- **Mock Data:** Simulated RFID scans and GPS coordinates
- **Hardware Testing:** Real Pi devices on test buses
- **Network Testing:** Connectivity and failover scenarios

13 Future Enhancements

13.1 Planned Features

1. **Real-time Updates:** WebSocket integration for live data
2. **Mobile Apps:** Native iOS/Android applications
3. **Analytics Dashboard:** Attendance trends and insights
4. **Geofencing:** Automated alerts when bus enters/exits zones
5. **Multi-school Support:** Multi-tenancy architecture
6. **Parent Mobile App:** Push notifications and live tracking
7. **SMS Notifications:** Alternative to in-app notifications
8. **Advanced Reporting:** Exportable reports (PDF, Excel)
9. **Offline Queue:** Retry failed uploads when connectivity restored
10. **Video Streaming:** Live camera feed from buses

13.2 Technical Improvements

1. **Microservices:** Split monolith into services
2. **Redis Cache:** Distributed caching layer
3. **Message Queue:** RabbitMQ/Kafka for async processing
4. **Load Balancing:** Multiple backend instances
5. **CDN Integration:** Photo delivery optimization
6. **Elasticsearch:** Advanced search capabilities
7. **GraphQL API:** Alternative to REST for flexible queries
8. **OAuth2:** Third-party authentication support

13.3 Security Enhancements

1. **Two-Factor Authentication:** OTP via SMS/email
2. **Audit Logs:** Comprehensive activity logging
3. **Encryption at Rest:** Database encryption
4. **Backup Encryption:** Encrypted backup files
5. **Rate Limiting:** API rate limiting per user/device
6. **GDPR Compliance:** Data privacy features

14 Appendices

14.1 Appendix A: Database Indexes

```

1 # Users collection
2 db.users.create_index("email", unique=True)
3
4 # Students collection
5 db.students.create_index([
6     ("class_name", 1),
7     ("section", 1),
8     ("roll_number", 1)
9 ], unique=True, name="unique_class_section_roll")
10
11 # Attendance collection
12 db.attendance.create_index([
13     ("student_id", 1),
14     ("date", -1)
15 ])
16
17 # Notifications collection
18 db.notifications.create_index([
19     ("user_id", 1),
20     ("created_at", -1)
21 ])
22
23 # Device keys collection
24 db.device_keys.create_index("bus_id", unique=True)

```

14.2 Appendix B: API Status Codes

Code	Status	Usage
200	OK	Successful GET, PUT, DELETE
201	Created	Successful POST
400	Bad Request	Invalid input data
401	Unauthorized	Missing/invalid credentials
403	Forbidden	Insufficient permissions
404	Not Found	Resource does not exist
409	Conflict	Duplicate or dependency violation
422	Unprocessable Entity	Validation error
500	Internal Server Error	Server-side error

14.3 Appendix C: Useful Commands

```

1 # Database operations
2 mongodump --db bus_tracker --out /backups/manual
3 mongorestore --db bus_tracker /backups/manual/bus_tracker
4
5 # Service management
6 sudo supervisorctl status
7 sudo supervisorctl restart all

```

```
8 sudo supervisorctl tail backend
9 sudo supervisorctl tail frontend
10
11 # Logs
12 tail -f /var/log/supervisor/backend.out.log
13 tail -f /var/log/supervisor/backend.err.log
14 journalctl -u bus-tracker-pi -f
15
16 # Pi testing
17 cd /app/tests
18 python3 pi_server.py
```

14.4 Appendix D: Dependencies

14.4.1 Backend (requirements.txt)

```
1 fastapi==0.104.1
2 uvicorn[standard]==0.24.0
3 motor==3.3.1
4 python-dotenv==1.0.0
5 bcrypt==4.0.1
6 pydantic==2.4.2
7 aiohttp==3.0.1
8 deepface==0.0.79
9 opencv-python==4.8.1.78
10 numpy==1.24.3
11 Pillow==10.1.0
```

14.4.2 Frontend (package.json highlights)

```
1 {
2   "dependencies": {
3     "react": "^18.2.0",
4     "react-dom": "^18.2.0",
5     "leaflet": "^1.9.4",
6     "react-leaflet": "^4.2.1",
7     "@radix-ui/react-dialog": "^1.0.5",
8     "@radix-ui/react-tabs": "^1.0.4",
9     "lucide-react": "^0.263.1",
10    "axios": "^1.5.1"
11  }
12 }
```

15 Glossary

API Application Programming Interface

ASGI Asynchronous Server Gateway Interface

bcrypt Password hashing algorithm

CORS Cross-Origin Resource Sharing

CRUD Create, Read, Update, Delete operations

DeepFace Deep learning face recognition library

FastAPI Modern Python web framework

GPIO General Purpose Input/Output (Pi pins)

GPS Global Positioning System

GPRS General Packet Radio Service

GSM Global System for Mobile Communications

HTTP Hypertext Transfer Protocol

IoT Internet of Things

JWT JSON Web Token

MongoDB NoSQL document database

Motor Async MongoDB driver for Python

MQTT Message Queuing Telemetry Transport

Pi Raspberry Pi single-board computer

RBAC Role-Based Access Control

REST Representational State Transfer

RFID Radio-Frequency Identification

SHA256 Secure Hash Algorithm 256-bit

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver-Transmitter

UUID Universally Unique Identifier

16 Conclusion

The Bus Tracker System represents a comprehensive solution for modern school transportation management, combining web technologies, IoT devices, and artificial intelligence to enhance student safety and operational efficiency.

16.1 Key Achievements

- **Full-Stack Integration:** Seamless connection between React frontend, FastAPI backend, and MongoDB database
- **IoT Innovation:** Raspberry Pi devices with RFID and face recognition
- **User-Centric Design:** Role-specific dashboards with intuitive interfaces
- **Production Ready:** Backup system, security features, and monitoring
- **Scalable Architecture:** Designed for growth and future enhancements

16.2 System Benefits

- **For Schools:** Reduced manual work, better oversight, data-driven decisions
- **For Parents:** Peace of mind, real-time updates, attendance visibility
- **For Teachers:** Efficient student tracking, attendance monitoring
- **For Administrators:** Complete system control, comprehensive management tools

16.3 Technology Highlights

The system leverages modern technologies:

- Async Python for high-performance backend
- React for responsive, dynamic user interfaces
- MongoDB for flexible, scalable data storage
- DeepFace for accurate face verification
- Leaflet for interactive map visualization
- Kubernetes for container orchestration

Contact Information

For technical support, questions, or contributions:

Documentation: /docs/

Repository: github.com/SiddharthJain131/bus-tracker

API Docs: /docs/API_DOCUMENTATION.md