Microservice Architecture - Documentation

Overview

This microservice architecture consists of two main backend services designed to provide a complete user management and notification system with role-based access control, two-factor authentication, and real-time communication capabilities.

Architecture Components

1. User Management Service

- **Port**: 3000 (default)
- **Purpose**: Core authentication, user management, organization management, and invitation system
- **Database**: MongoDB
- **Key Features**: JWT authentication, 2FA (OTP/TOTP), role-based access control, invitation workflow

2. Notifications Service

- **Port**: 4000 (default)
- **Purpose**: Email notifications and real-time communication via WebSockets
- **Key Features**: Email sending, RabbitMQ integration, Socket.IO server with Redis adapter

Service Details

User Management Service

Technology Stack

- **Runtime**: Node.js with ES modules
- **Framework**: Express.js
- **Database**: MongoDB with Mongoose ODM
- **Authentication**: JWT with refresh token rotation
- **Security**: Helmet, CORS, rate limiting, bcrypt password hashing
- **2FA**: Email OTP and TOTP (Google Authenticator compatible)
- **Communication**: RabbitMQ for event publishing, Socket.IO client
- **Documentation**: Swagger/OpenAPI integration

Core Models

User Model

```
{
  email: String (unique, indexed),
  password: String (bcrypt hashed),
  firstName: String,
  lastName: String,
  role: Enum [super_admin, site_admin, operator, client_admin,
client_user],
  twoFactorMethod: Enum [otp, totp],
  totpSecret: String (encrypted),
  isTotpEnabled: Boolean,
  organization: ObjectId (ref: Organization),
  invitedBy: ObjectId (ref: User),
  isActive: Boolean,
  lastLogin: Date,
  createdAt: Date,
  updatedAt: Date
}
```

Organization Model

```
{
  name: String,
  slug: String (unique, indexed),
  twoFactorMethod: Enum [otp, totp] (default: otp),
  adminUser: ObjectId (ref: User),
  isActive: Boolean,
  createdAt: Date,
  updatedAt: Date
}
```

Invite Model

```
{
  email: String (indexed),
  role: Enum [site_admin, operator, client_admin, client_user],
  invitedBy: ObjectId (ref: User),
  organization: ObjectId (ref: Organization),
  organizationName: String,
  token: String (unique, indexed),
  status: Enum [pending, accepted, expired, revoked],
  expiresAt: Date,
  acceptedAt: Date,
  createdAt: Date
}
```

RefreshToken Model

```
{
  token: String (unique, indexed),
  userId: ObjectId (ref: User),
  expiresAt: Date,
  createdAt: Date
}
```

API Routes

Authentication Routes (`/api/auth`)

Method	Endpoint	Description	Auth Required	Roles
POST	`/login`	User login with email/password	No	-
POST	`/verify-otp`	Verify email OTP code	No	-
POST	`/verify-totp`	Verify TOTP from authenticator app	No	-
POST	`/refresh`	Refresh access token	No	-
POST	`/logout`	Logout and revoke refresh token	No	-
GET	`/profile`	Get authenticated user profile	Yes	All
POST	`/totp/setup`	Generate QR code for TOTP setup	Yes	All
POST	`/totp/confirm`	Confirm TOTP setup with verification	No	-
POST	`/mfa/change`	Change MFA method (OTP ↔ TOTP)	Yes	super_admin, site_admin, operator, client_admin

Login Flow Examples:

1. **Login without 2FA:**

```
POST /api/auth/login
{
  "email": "user@example.com",
  "password": "password123"
}
Response: {
  "success": true,
  "accessToken": "jwt_token",
  "refreshToken": "refresh_token",
  "user": { ... }
}
```

2. **Login with OTP 2FA:**

```
POST /api/auth/login
Response: {
  "success": true,
  "requiresTwoFactor": true,
  "twoFactorMethod": "otp",
  "userId": "user_id"
}

POST /api/auth/verify-otp
{
  "userId": "user_id",
  "otp": "123456"
}
```

3. **Login with TOTP 2FA:**

```
POST /api/auth/login
Response: {
  "success": true,
  "requiresTwoFactor": true,
  "twoFactorMethod": "totp",
  "userId": "user_id"
}

POST /api/auth/verify-totp
{
  "userId": "user_id",
  "token": "123456"
}
```

Invitation Routes (`/api/invites`)

Method	Endpoint	Description	Auth Required	Roles
POST	`/create`	Create user invitation	Yes	super_admin, site_admin, operator, client_admin
POST	`/accept`	Accept invitation and create account	No	-
GET	`/details/:token`	Get invitation details	No	-
GET	`/list`	List all invitations	Yes	super_admin, site_admin, operator, client_admin

Method	Endpoint	Description	Auth Required	Roles
DELETE	`/:inviteId/revoke`	Revoke pending invitation	Yes	super_admin, site_admin, operator, client_admin

Invitation Flow:

1. **Create Invitation:**

```
POST /api/invites/create
{
  "email": "newuser@example.com",
  "role": "client_user",
  "organizationName": "Acme Corp"
}
```

2. **Accept Invitation:**

```
POST /api/invites/accept
{
  "token": "invite_token",
  "firstName": "John",
  "lastName": "Doe",
  "password": "SecurePass123",
  "twoFactorMethod": "otp"
}
```

Organization Routes (`/api/organization`)

Method	Endpoint	Description	Auth Required	Roles
GET	`/`	Get organization details	Yes	client_admin, client_user
PUT	`/`	Update organization	Yes	client_admin
GET	`/members`	Get organization members	Yes	client_admin, client_user

Role-Based Access Control

Role Hierarchy

1. **super_admin** - System-wide administrative access
2. **site_admin** - Site-level administrative access
3. **operator** - Operational access with invite creation rights
4. **client_admin** - Organization administrative access
5. **client_user** - Standard user access within organization

Permission Matrix

Action	super_admin	site_admin	operator	client_admin	client_user
Create Invites	✅	✅	✅	✅	❌
Change MFA Method	✅	✅	✅	✅	❌
Update Organization	❌	❌	❌	✅	❌
View Organization	❌	❌	❌	✅	✅
View Members	❌	❌	❌	✅	✅

Two-Factor Authentication

MFA Methods

1. **OTP (One-Time Password)**: 6-digit codes sent via email
2. **TOTP (Time-based One-Time Password)**: Google Authenticator compatible

MFA Rules

- **Client Users**: Always inherit organization's MFA method, cannot change individually
- **Admins**: Can change their MFA method via `/api/auth/mfa/change`
- **Organization Admin**: Can set organization-wide MFA method

TOTP Setup Flow

1. User requests TOTP setup via `/api/auth/totp/setup`
2. System generates QR code and secret
3. User scans QR code with authenticator app
4. User verifies with 6-digit code via `/api/auth/totp/confirm`
5. TOTP is enabled for future logins

Security Features

Rate Limiting

- **General API**: 100 requests per 15 minutes
- **Authentication**: 50 requests per 15 minutes
- **Login attempts**: Tracks failed attempts

Token Management

- **Access Token**: 15-minute expiry
- **Refresh Token**: 7-day expiry with rotation
- **Invite Token**: 7-day expiry
- **OTP**: 10-minute expiry

Password Security

- **Hashing**: bcrypt with 12 salt rounds
- **Validation**: Minimum 8 characters
- **OTP Hashing**: bcrypt with 10 salt rounds

Notifications Service

Technology Stack

- **Runtime**: Node.js with ES modules
- **Framework**: Express.js
- **Real-time**: Socket.IO with Redis adapter
- **Email**: Nodemailer with SMTP support
- **Message Queue**: RabbitMQ for event consumption
- **Caching**: Redis for Socket.IO scaling

Core Features

Email System

- **SMTP Integration**: Configurable SMTP server support
- **Template Support**: HTML and text email templates
- **Fallback**: Ethereal email for development
- **Validation**: Email address and content validation

Real-time Communication

- **Socket.IO Server**: WebSocket connections with fallback to polling
- **Redis Adapter**: Multi-instance scaling support
- **Event Handling**: User registration, invite acceptance notifications
- **Room Management**: User-specific notification rooms

Message Queue Integration

- **RabbitMQ Consumer**: Processes email events from User Management service
- **Event Types**: Invite creation, user registration, system notifications
- **Error Handling**: Dead letter queues and retry mechanisms

API Routes

Email Routes (`/api/email`)

Method	Endpoint	Description	Auth Required
POST	`/send`	Send email notification	No

Email Request Example:

```
POST /api/email/send
{
  "to": "user@example.com",
  "subject": "Welcome to the System",
  "html": "<h1>Welcome!</h1><p>Your account has been created.</p>",
  "text": "Welcome! Your account has been created."
}
```

Socket.IO Events

Client Events

- **register**: Register user for notifications

  ```
  socket.emit('register', { userId: 'user_id' });
  ```

- **inviteAccepted**: Notify about invite acceptance

  ```
  socket.emit('inviteAccepted', {
    userId: 'user_id',
    message: 'User accepted invitation',
    timestamp: new Date().toISOString()
  });
  ```

Server Events

- **notification**: User-specific notifications

  ```
  socket.on('notification', (data) => {
    // Handle notification
  });
  ```

- **inviteStatusUpdate**: Global invite status updates

  ```
  socket.on('inviteStatusUpdate', (data) => {
    // Handle invite status change
  });
  ```

Configuration

Environment Variables

```
# RabbitMQ
RABBITMQ_URL=amqp://localhost:5672
```

```
RABBITMQ_EXCHANGE=events
RABBITMQ_QUEUE_EMAIL=notifications.email
RABBITMQ_ROUTE_INVITE=user.invite.created

# Redis
REDIS_URL=redis://localhost:6379

# SMTP
SMTP_HOST=smtp.example.com
SMTP_PORT=587
SMTP_USER=apikey
SMTP_PASS=secret
SMTP_FROM="System <no-reply@example.com>"

# CORS
CORS_ORIGIN=http://localhost:5173
```

System Integration

Inter-Service Communication

RabbitMQ Events

The User Management service publishes events that the Notifications service consumes:

1. **User Invite Created**
   - **Event**: `user.invite.created`
   - **Payload**: `{ to, subject, html, text }`
   - **Action**: Send invitation email
2. **User Registration**
   - **Event**: `user.registered`
   - **Payload**: `{ to, subject, html, text }`
   - **Action**: Send welcome email

Socket.IO Integration

- User Management service connects to Notifications service as a client
- Publishes real-time events for invite acceptance
- Enables live updates in the frontend application

Database Schema Relationships

```
User (1) ←→ (1) Organization
User (1) ←→ (N) Invite (as inviter)
User (1) ←→ (N) RefreshToken
Organization (1) ←→ (N) User (as members)
Invite (N) ←→ (1) Organization
```

Health Checks

User Management Service

```
GET /health
Response: {
  "success": true,
  "message": "User Management Service is running",
  "timestamp": "2024-01-01T00:00:00.000Z"
}
```

Notifications Service

```
GET /health
Response: {
  "success": true,
  "message": "Notifications Service is running",
  "timestamp": "2024-01-01T00:00:00.000Z"
}
```

Development Setup

Prerequisites

- Node.js 18+
- MongoDB 6+
- Redis 6+
- RabbitMQ 3.8+

Environment Setup

User Management Service

```
cd userManagement
npm install
cp .env.example .env
# Configure environment variables
npm run dev
```

Notifications Service

```
cd notifications
npm install
cp .env.example .env
# Configure environment variables
npm run dev
```

Environment Variables

User Management Service (.env)

```
# Database
MONGODB_URI=mongodb://localhost:27017/user_management

# JWT
JWT_SECRET=your-super-secret-jwt-key
JWT_REFRESH_SECRET=your-super-secret-refresh-key

# CORS
CORS_ORIGIN=http://localhost:5173

# RabbitMQ
RABBITMQ_URL=amqp://localhost:5672

# Notifications Service
NOTIFICATIONS_SERVICE_URL=http://localhost:4000
```

Notifications Service (.env)

```
# RabbitMQ
RABBITMQ_URL=amqp://localhost:5672
RABBITMQ_EXCHANGE=events
RABBITMQ_QUEUE_EMAIL=notifications.email
RABBITMQ_ROUTE_INVITE=user.invite.created

# Redis
REDIS_URL=redis://localhost:6379

# SMTP
SMTP_HOST=smtp.example.com
SMTP_PORT=587
SMTP_USER=your-smtp-user
SMTP_PASS=your-smtp-password
SMTP_FROM="System <no-reply@example.com>"

# CORS
CORS_ORIGIN=http://localhost:5173
```

API Documentation

Swagger Integration

The User Management service includes comprehensive Swagger/OpenAPI documentation:

- **Access**: `http://localhost:3000/api-docs`
- **JSON Spec**: `http://localhost:3000/api-docs.json`
- **Features**: Interactive testing, authentication, request/response examples

API Response Format

Success Response

```
{
  "success": true,
  "data": { ... },
  "message": "Operation successful"
}
```

Error Response

```
{
  "success": false,
  "message": "Error description",
  "errors": [ ... ] // Optional validation errors
}
```

HTTP Status Codes

- **200**: OK
- **201**: Created
- **400**: Bad Request
- **401**: Unauthorized
- **403**: Forbidden
- **404**: Not Found
- **500**: Internal Server Error

Production Considerations

Security

- Use HTTPS in production
- Rotate JWT secrets regularly
- Implement proper CORS policies
- Use environment-specific rate limits

- Enable request logging and monitoring

Scalability

- Use Redis for session storage
- Implement database connection pooling
- Use load balancers for multiple service instances
- Configure RabbitMQ clustering for high availability

Monitoring

- Implement health check endpoints
- Use structured logging (Winston, Pino)
- Set up application performance monitoring
- Monitor database and message queue performance

Deployment

- Use containerization (Docker)
- Implement CI/CD pipelines
- Use environment-specific configurations
- Set up automated backups for databases

Troubleshooting

Common Issues

Database Connection

- Verify MongoDB is running and accessible
- Check connection string format
- Ensure database user has proper permissions

RabbitMQ Issues

- Verify RabbitMQ is running
- Check exchange and queue configurations
- Monitor message queue for dead letters

Authentication Problems

- Verify JWT secrets are consistent
- Check token expiration times
- Ensure proper CORS configuration

Email Delivery

- Verify SMTP credentials
- Check email provider rate limits

- Monitor email delivery logs

Logging

Both services use structured logging with different levels:

- **ERROR**: System errors and exceptions
- **WARN**: Warning conditions
- **INFO**: General information
- **DEBUG**: Detailed debugging information

Support and Maintenance

Code Structure

- **Controllers**: Handle HTTP requests and responses
- **Services**: Business logic and data processing
- **Models**: Database schemas and validation
- **Middleware**: Authentication, validation, error handling
- **Utils**: Helper functions and utilities
- **Config**: Service configuration and constants

Testing

- Unit tests for business logic
- Integration tests for API endpoints
- End-to-end tests for complete workflows
- Load testing for performance validation

Documentation Updates

- Keep API documentation current
- Update environment variable documentation
- Maintain deployment guides
- Document any breaking changes