Echelon – Employee Hierarchy Management System

Technical Documentation

Qwinton Knocklein

Github Link: https://github.com/Cyb3rC0de7/echelon

Website Link: https://echelon-epiuse-8bc39b155203.herokuapp.com/

Table of Contents

System Overview

Echelon is a cloud-hosted employee hierarchy management system built with a modern, full-stack architecture. The application provides comprehensive CRUD operations for employee data, visual hierarchy representation, and role-based access control with four permission levels (admin, hr, manager, employee)

Key Features

- Employee management with full CRUD operations
- Visual hierarchy tree with drag-and-drop functionality
- Role-based permission system
- Gravatar integration for profile pictures
- Advanced search and filtering
- Password management system
- Responsive Material UI interface

Architecture

Overall Architecture Pattern: Three-Tier Architecture

The system follows a classic three-tier architecture pattern:

1. **Presentation Tier:** React.js frontend with Material-UI
2. **Business Logic Tier:** Node.js/Express.js REST API
3. **Data Tier:** PostgreSQL database

Component Architecture

Backend Components

- **Controllers:** Route handlers in */routes* directory
- **Models:** Sequelize ORM model for database entity
- **Middleware:** Authentication and authorization layers.
- **Services:** Business logic separation (implied by route handlers)

Frontend Components

- **Context Providers:** Authentication state management
- **UI Components:** Reusable React components
- **Services:** API communication layer
- **State Management:** React hooks and context

Technology Stack

Frontend Technologies

- **React:** Modern functional components with hooks and context
- **Material-UI:** Component library for consistent UI
- **React DnD:** Drag-and-Drop functionality for hierarchy management
- **D3.js:** Tree visualization and layout calculations
- **Axios:** HTTP client for API communication
- **Day.js:** Date manipulation and formatting

Backend Technologies

- **Node.js:** JavaScript runtime environment
- **Express.js:** Web application framework
- **Sequelize ORM:** Database abstraction and modeling
- **PostgreSQL:** Relational database management system
- **JWT:** JSON Web Tokens for authentication
- **Bcrypt:** Password hashing and verification
- **CORS:** Cross-Origin Resource Sharing middleware

Development and Deployment

- **Git:** Version Control system
- **Heroku:** Cloud platform for deployment
- **Dotenv:** Environment variable management
- **MD5:** Frontend Gravatar hash generation

Database Design

Entity Relationship Model

The system uses a self-referencing Employee entity with the following schema:

```sql
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    employeeNumber VARCHAR UNIQUE NOT NULL,
    name VARCHAR NOT NULL,
    surname VARCHAR NOT NULL,
    email VARCHAR UNIQUE NOT NULL,
    birthDate DATE NOT NULL,
    salary DECIMAL(10,2) NOT NULL,
    role VARCHAR NOT NULL,
    password VARCHAR NOT NULL,
    requiresPasswordReset BOOLEAN DEFAULT true,
    permissionLevel ENUM('employee', 'manager', 'hr', 'admin') DEFAULT 'employee',
    isActive BOOLEAN DEFAULT true,
    managerId INTEGER REFERENCES employees(id),
    createdAt TIMESTAMP DEFAULT NOW(),
    updatedAt TIMESTAMP DEFAULT NOW()
);
```

Relationships

- **Self-referencing Hierarchy:** Each employee can have a manager (*managerId* foreign key)
- **One-to-Many:** Manager → Subordinates relationship
- **Constraints:** Employees cannot be their own manager (enforced in application logic)

Indexes

- Primary key on *id*
- Unique indexes on *employeeNumber* and *email*
- Foreign key index on *managerId* for hierarchy queries

Security Implementation

Authentication Strategy

- JWT-Based authentication with 24-hour token expiration
- Bcrypt password hashing with 12 rounds of salt
- Secure token storage in *localStorage* (client-side)

Authorization levels

1. **Admin:** Full system access, can manage all employees and permissions
2. **HR:** Can manage all non-admin employees, cannot assign admin privileges
3. **Manager:** Can manage direct subordinates
4. **Employee:** Can view and edit their own profile, as well as view colleagues with the same manager.

Security Measures

- Password complexity requirements (minimum 6 characters)
- Default password generation for new employees (*nameEmployeeNumber*)
- Token-based session management
- Input validation and sanitization
- SQL injection prevention through ORM

API Design

RESTful API Structure

Authentication Endpoints

```
  // POST api/auth/login
> router.post('/login', async (req, res) => {···
  });

  //Change password
> router.put('/change-password', authenticateToken, async (req, res) => {···
  });

  // Reset password to default (admin only)
> router.put('/reset-password/:employeeId', authenticateToken, async (req, res) => {···
  });

  // Get current user
> router.get('/me', authenticateToken, async (req, res) => {···
  });

  // Logout (client-side token removal)
> router.post('/logout', authenticateToken, (req, res) => {···
  });
```

Employee Management

```
  // GET /api/employees - Get all employees with optional search and filters and permission filtering
> router.get('/', authenticateToken, async (req, res) => {···
  });

  // GET /api/employees/:id - Get single employee
> router.get('/:id', authenticateToken, async (req, res) => {···
  });

  // POST /api/employees - Create new employee
> router.post('/', authenticateToken, async (req, res) => {···
  });

  // PUT /api/employees/:id - Update employee
> router.put('/:id', authenticateToken, async (req, res) => {···
  });

  // PUT /api/employees/:id/manager - Update employee's manager
> router.put('/:id/manager', authenticateToken, async (req, res) => {···
  });

  // DELETE /api/employees/:id - Delete employee
> router.delete('/:id', authenticateToken, async (req, res) => {···
  });

  // GET /api/employees/hierarchy - Get full organization hierarchy
> router.get('/hierarchy/tree', authenticateToken, async (req, res) => {···
  });
```

Admin Operations

```
11   // Get system statistics
12 > router.get('/stats', async (req, res) => {···
42   });
43
44   // Bulk permission updates
45 > router.put('/bulk-permissions', async (req, res) => {···
60   });
61
62   // Export employee data
63 > router.get('/export/:format', async (req, res) => {···
91   });
92
93   // System health check
94 > router.get('/health', async (req, res) => {···
112  });
```

Response Format

```
{
  "data": {},
  "message": "Success Message",
  "error": "Error Message"
}
```

Frontend Implementation

State Management Strategy

- React Context API for global authentication state
- Local component state (*useState*) for component-specific data
- Effect hooks (*useEffect*) for side effects and data fetching

Component Architecture Pattern: Container/Presentational

- **Container Components:** Handle data fetching and business logic
- **Presentational Components:** Focus on UI rendering and user interaction

Key Components

1. **App.jsx:** Main application router and authentication wrapper
2. **EmployeeList.jsx:** Data table with filtering and CRUD operations
3. **EmployeeForm.jsx:** Form component with role-based field permissions
4. **HierarchyView.jsx:** Interactive tree visualization with D3.js
5. **Login.jsx:** Authentication interface

UI/UX Patterns

- **Material Design:** Consistent visual language
- **Progressive Enhancement:** Features unlock based on user permissions
- **Responsive Design:** Mobile-friendly interface
- **Loading States:** User feedback during async operations

Deployment Strategy

Cloud Deployment on Heroku

Production Configuration

- **Environment Variables:** Sensitive data stored in Heroku config vars
- **Database:** Heroku Postgres add-on with SSL
- **Build Process:** Automated deployment from Git repository
- **Static Assets:** React build served by Express in production mode

Environment Management

```
34  // Serve static files from React build (for production)
35  if (process.env.NODE_ENV === 'production') {
36    const buildPath = path.join(__dirname, '../../client/build');
37
38    console.log('Serving static files from:', buildPath);
39
40    // Serve static files
41    app.use(express.static(buildPath));
42
43    // Handle React routing - send all non-API requests to index.html
44    app.get('/', (req, res) => {
45      res.sendFile(path.join(buildPath, 'index.html'));
46    });
47  } else {
48    app.get('/', (req, res) => {
49      res.json({ message: 'Echelon API - Development Mode' });
50    });
51  }
```

Design Patterns

1. MVC (Model-View-Controller)

- **Models:** Sequelize entities (*Employee*)
- **Views:** React components
- **Controllers:** Express route handlers

2. Repository Pattern (Implicit)

- Sequelize ORM acts as repository abstraction
- Database operations encapsulated in model methods

3. Middleware Chain Pattern

- Authentication middleware (*authenticateToken*)
- Authorization middleware (*requirePermission*)
- Error handling middleware

4. Factory Pattern

- JWT token generation
- Default password creation
- Gravatar URL generation

5. Observer Pattern

- React's *useEffect* hooks for state changes
- Event-driven UI updates

6. Strategy Pattern

- Different permission strategies based on user role
- Conditional rendering based on user context

Justifications

Technology Choices

Why React?

- **Component Reusability:** Modular architecture with reusable UI components
- **Virtual DOM:** Efficient rendering for complex hierarchy visualizations
- **Rich Ecosystem:** Extensive library support (Material-UI, D3, React DnD)
- **Developer Experience:** Excellent debugging tools and documentation

Why Node.js/Express?

- **JavaScript Ecosystem:** Full-stack JavaScript reduces context switching
- **NPM Ecosystem:** Rich package availability for rapid development
- **Asynchronous I/O:** Excellent performance for concurrent requests
- **REST API Simplicity:** Express provides a minimal, flexible web framework

Why PostgreSQL?

- **ACID Compliance:** Ensures data integrity for employee records
- **Hierarchical Queries:** Efficient support for recursive relationships
- **JSON Support:** Flexibility for future schema evolution
- **Heroku Integration:** Native support and easy deployment

Why JWT Authentication?

- **Stateless:** No server-side session storage required
- **Scalability:** Easy to scale horizontally without session affinity
- **Cross-Platform:** Works across web, mobile, and API clients
- **Security:** Encrypted token with expiration management

Architectural Decisions

Three-Tier Architecture

- **Separation of Concerns:** Clear boundaries between presentation, business, and data layers
- **Scalability:** Each tier can be scaled independently
- **Maintainability:** Changes in one layer don't cascade to others
- **Testing:** Each layer can be tested in isolation

Role-Based Access Control

- **Security:** Prevents unauthorized access to sensitive data
- **Compliance:** Supports organizational policies and audit requirements
- **Flexibility:** Easy to add new roles or modify permissions
- **User Experience:** Progressive feature disclosure based on user privileges

Single-Page Application (SPA)

- **User Experience:** Smooth, responsive interface without page reloads
- **Performance:** Reduced server load and faster navigation
- **Modern Standards:** Aligns with current web development best practices
- **Mobile Compatibility:** Better mobile experience than traditional multi-page apps

Performance Optimizations

Frontend

- **Lazy Loading:** Code splitting for route-based chunks
- **Virtual Scrolling:** Efficient rendering of large employee lists
- **Debounced Search:** Reduced API calls during user input

Backend

- **Database Indexing:** Optimized queries for employee lookups
- **Eager Loading:** Includes related data to reduce N+1 queries
- **Connection Pooling:** Sequelize manages database connection efficiency
- **Middleware Optimization:** Early authentication checks to prevent unnecessary processing

Security Considerations

Authentication Security

- **Password Hashing:** bcrypt with high salt rounds prevents rainbow table attacks
- **Token Expiration:** 24-hour JWT expiration reduces exposure window
- **HTTPS Only:** Secure transmission of authentication tokens
- **Input Validation:** Prevents injection attacks and data corruption

Authorization Security

- **Principle of Least Privilege:** Users only see data they need
- **Role-Based Permissions:** Granular control over feature access
- **Server-Side Validation:** All permissions enforced on backend
- **Audit Trail:** Database timestamps track all changes

Conclusion

The Echelon employee hierarchy management system demonstrates modern full-stack development practices with a focus on security, scalability, and user experience. The chosen technologies and architectural patterns provide a solid foundation for future enhancements while maintaining code quality and system reliability.

The role-based permission system ensures appropriate access control, while the visual hierarchy component provides an intuitive interface for organizational management. The cloud deployment strategy ensures high availability and easy maintenance.