EMS TECHNICAL DOCUMENT

Cameron Taberer

Contents

Technical Document for EPI-USE Africa Employee Management System (EMS)

Hosted Web Site: https://employeemanagementprogrameta.azurewebsites.net/

GitHub: https://github.com/CameronTaberer/Employee_Management_Program

1. Employee Management Application

1.1 Overview

In the development of the cloud-hosted application for managing EPI-USE Africa's employee hierarchy, a variety of technologies were selected to address the specific requirements and challenges of the project. This document outlines the rationale behind these technology choices, the architecture of the data structure, and the approach to coding in a modular fashion. Additionally, it delves into the implementation of specific functionalities, such as the advanced filter functions.

1,2 Technology Stack and Justifications

1.2.1 Angular for the Frontend

Angular was chosen for its robust ecosystem and comprehensive support for building dynamic, responsive user interfaces. It is very powerful and the front-end framework I have the most experience in. Its data-binding capabilities, adherence to the MVC pattern, and extensive libraries facilitated the development of this application that efficiently manages and display's data structures like the organizational hierarchy.

1.2.2 .NET 8 for Backend Development

.NET 8 offered a strong, scalable platform for backend development, aligning with Angular for a seamless full-stack solution. I also have a lot of experience in this framework. Its support for strong typing and extensive libraries for web APIs also nice to have. Moreover, .NET 8's integration capabilities with Angular and Azure streamlined the deployment process, enhancing the application's accessibility and security.

1.2.3. SQL Server Management Studio (SSMS)

SSMS was selected for its powerful data management capabilities, essential for handling the complex queries and relational data structures within the project. Its reliability and seamless integration with .NET applications ensured efficient data manipulation and integrity, vital for the robust performance of the application.

1.2.4 Deployment on Azure

Azure provided a scalable and reliable cloud platform that complemented the application's requirements. Its range of services and ease of integration with the chosen technology stack made it the ideal choice for hosting, ensuring that the application was robust, scalable, and easily accessible to end-users.

1.3 Architecture and Data Structure

The application was designed with a modular architecture, dividing the codebase into distinct layers: API, Services, and Components. This approach enhanced the maintainability and scalability of the application, simplifying error handling and allowing for easier future enhancements.

The data models (Employee, Position, EmployeeViewModel) were meticulously designed to accurately represent the organization's structure while avoiding potential pitfalls such as circular references, particularly with the Manager_ID attribute in the Employee model. This ensured the integrity of the data structure and facilitated the management of complex hierarchical relationships.

The enterprise architecture for the Employee Management Program API is designed around a set of models that represent the core entities within the organizational structure: Employees and Positions. These models facilitate the interaction with the underlying database and serve as the foundation for data manipulation and business logic within the application. Here's a basic description of each component and how they interrelate:

Employee Model

The **Employee** model is the central entity representing an individual staff member within the organization. It includes several key attributes:

- **Employee_ID**: A unique identifier for each employee, serving as the primary key in the database.
- **Employee_Name** and **Employee_Surname**: Required fields to store the employee's name and surname.
- **Employee_Email**: A required field for the employee's email address, which must be unique to ensure that each employee can be distinctly identified and contacted.
- **GravatarUrl**: An optional field to store the URL of the employee's Gravatar profile picture.
- **Birth_Date**: The employee's date of birth, a required piece of information that may be used for demographic analysis or birthday acknowledgments.
- **Salary**: A required decimal field to store the employee's salary, reflecting their compensation within the organization.
- **Position_ID**: A foreign key that relates the **Employee** to a **Position**, indicating the role they occupy in the hierarchy.

- **Manager_ID**: Another integer key that refers to the **Employee** who is this individual's line manager, establishing the reporting chain within the organization. It is required to maintain the structure of reporting relationships.

The **Employee** model also defines a navigation property, **Position**, which facilitates an object-relational mapping to the **Position** entity, enabling direct access to the associated position details.

Position Model

The **Position** model represents the different roles within the organization and includes:

- **Position_ID**: A unique identifier for each position, acting as the primary key.
- **Position_Name**: The name of the position, which is a required field that describes the role (e.g., "Software Developer").
- **Hierarchy_Level**: A required integer that establishes the position's level within the organizational hierarchy, with a lower number representing a higher-ranking position.

This model underpins the organizational structure and allows the system to enforce hierarchical logic within business operations, such as assigning managers to employees.

EmployeeViewModel

The **EmployeeViewModel** is a data transfer object (DTO) that represents a more secure or customized form of the **Employee** entity for data exchange purposes. This model omits certain attributes that may be sensitive or not needed when transferring data between the server and client. It includes similar attributes to the **Employee** model but can be extended or altered to suit the needs of different views or operations without affecting the database schema directly.

Architecture Summary

The enterprise architecture, represented by these models, showcases a clean separation between the database structure and the application's business logic. It allows the API to handle CRUD (Create, Read, Update, Delete) operations and present data to the client in a structured and secure manner. The architecture supports the MVC (Model-View-Controller) pattern with a clear demarcation of concerns, promoting maintainability and scalability. The use of annotations like **[Key]** and **[Required]** further reinforces database integrity and validation at the model level.

1.3.1 Implementation Insights

The development process included the implementation of advanced filter functions (filterEmployees method), showcasing the application's ability to perform dynamic searches within the organizational hierarchy. This functionality enhanced the user experience by allowing efficient navigation and management of the employee data.

Additionally, the buildHierarchy and transformEmployeesToTreeNodes functions were developed to construct and visualize the organizational hierarchy effectively. These functions demonstrated the application's capability to handle and present complex data structures, making the organizational structure accessible and understandable to users.

1.4 Conclusion

This document reflects the thought process and rationale behind each decision made during the project's development.