Employee Management App Design Document

Overview

This document describes the full stack employee management application that I have created. This application uses Angular version 19.2.7 on the front end and .NET Core 6.0 on the backend.

Backend

The backend API consists of 4 main folders as well as our Program.cs file which runs the application.

Data Folder

This folder holds our database context class - AppDbContext. This class is the main class in Entity Framework Core that communicates with the database. It is a link between our model (entity) classes and our database. It is utilized for data queries and saves.

The components of the DbContext are:

DbSet<TEntity> (DbSet<Employee) - collections of entities in the context. Typically, there should be one DbSet<TEntity> for each entity class in a model.

```
namespace EmployeeManagement.Data
{
    public class AppDBContext : DbContext
    {
        public DbSet<Employee> Employees { get; set; }

        public AppDBContext(DbContextOptions<AppDBContext> options)
            : base(options)
        { }
    }
}
```

Configure data source - used to configure the database (and other options) to be used for our database context. It is added to Program.cs.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDBContext>(
    options => options.UseInMemoryDatabase("EmployeeDB")
    );
```

You need to make sure you have installed the required NuGet package base on your data source. For this application we are using an in-memory database.

Repositories Folder

This folder holds classes and interfaces dedicated to providing logic to access, store, retrieve and map data to domain objects. This allows us to separate concerns and keep the business logic away from data access logic which helps simplify and improve code maintainability.

The first component of our folder is the Repository Interface - IEmployeeRepository. This file establishes the functions that must be present in our concrete implementation. We usually define CRUD operations, but we could also extend our interface with more functions if more specific functionality is needed of our repository class.

The second component is the concrete repository - EmployeeRepository. This class implements the Repository Interface and maintains the actual code to interact with the data source - database, API, or InMemory collection. Through dependency injection, we inject our AppDbContext into the constructor of our concrete repository. This allows the repository to work with the correct context of our data source and perform the correct actions on the correct data.

Models

The models' folder holds our model classes in this Entity Framework Core design. The Entity Framework is an object-relational mapper for .NET applications supported by Microsoft. It allows developers to work with a database using .NET objects like our Employee object in our Employee.cs model class. Entity Framework Core is a lightweight expandable version of the Entity Framework that we use in this project.

To use this pattern first define a data model using a C# class. In this application we have our Employee Class. It gets and sets properties related to Employee information.

```
public class Employee
{
    public int Id { get; set; }

    [Required(ErrorMessage = "First Name is Required")]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "Last Name is Required")]
    public string LastName { get; set; }

    [Required(ErrorMessage = "Email is Required")]
    [EmailAddress(ErrorMessage = "Invalid Email Address")]
    public string Email { get; set; }

    [Required(ErrorMessage = "Phone is Required")]
    public string Phone { get; set; }

    [Required(ErrorMessage = "Position is Required")]
    public string Position { get; set; }
}
```

The second step is to create our DbContext class that will manage the model and handle our database connections.

We later add Data Annotations to provide validation for our Employee fields. This is a validation effort to ensure we only store valid employee information.

Properties

This folder holds the launch settings for our backend API. It specifies the main and backup port that our API uses when the API is running and other IIS information.

Program.cs

This is the main program file. In this file we build our web application, create a CORS policy so only our client host can hit our API, add our controllers and create a swagger page before running the application.

Frontend

The frontend application mainly consists of our Service class, our app components, our employee table component folder, our employee form folder and our bootstrap styling that we downloaded from bootstrap.

Services

The goal of our service file is to allow various application components to share functionality and data. They help with encapsulating data access, business logic and other non-view-related tasks. You can use *ng g s [service-name]* in VS Code command line to create a service if you have the Angular CLI installed.

A class can be marked as a service that is ready of injection by using the @Injectable decorator. The providedIn: 'root' indicates that the service is accessible from anywhere in the application and doesn't require explicit provisioning in any module. Our employee service class holds 5 methods to

perform CRUD operations for employees.

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { environment } from '../environments/environment';
import { Employee } from '../models/employee';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class EmployeeService {

  private apiUrl = `${environment.apiUrl}/employee`

  constructor(private httpClient: HttpClient) { }

  getEmployees(): Observable<Employee[]> {
    return this.httpClient.get<Employee[]>(this.apiUrl);
  }
```

We import provideHttpClient in our app.config.ts class to configure our app to know register that we will be using a httpClient. We later import an instance of our httpClient in our service and ingest it via dependency injection so that we can handle http requests. We can now use this service inside of other components to provide data to those components like our employee form and employee table components. This can also be used inside other services. Once a

service is injected into a component, you can use its methods and properties in that component.

```
export class EmployeeTableComponent {

  employees: Employee[] = [];

  constructor(private employeeService: EmployeeService, private router: Router){}
```

Observables

An Observable is analogous to a data stream to which you may subscribe. It is a component of the RxJS library, which facilitates the management of asynchronous tasks like user inputs and HTTP requests. Observables handle data that comes in the future like API responses. They manage events like user clicks or input changes. They also stream data continuously and provide real-time updates.

The HTTPClient service in Angular processes http requests and returns observables. Each of the functions in our service returns observables to be

ingested by the components that call them.

```
getEmployeeById(id: number): Observable<Employee> {
  return this.httpClient.get<Employee>(`${this.apiUrl}/${id}`);
}

createEmployee(employee: Employee): Observable<Employee> {
  return this.httpClient.post<Employee>(this.apiUrl, employee);
}
```

Components

In Angular components have a typescript file that is linked to a html file. It is different than React where you have jsx/tsx at the bottom of a file, but it is essentially the same concept. There are 3 components in this front end application, the app, employee table and employee form components. The app component consists of the main application. For us this will consist of our nav bar component and our router outlet which essentially encompasses the body of our app.

Employee Table Component

In our typescript file, we name the component selector so the component can be targeted if called in html by other components. We import the CommonModule to be able to use basic angular directives and pipes like ngFor, ngIf, etc, in the html of components that consume our component. The template is the html class that is associated with our component - employee-table.component.html. We finally have a CSS class for styling.

We import OnIt from angular/core so that we can initialize our component with whatever data is already in our in-memory database from our backend. We subscribe to the observable that is returned from our employee service and initialize our table with all our employee data that is present. We provide functionality for deleting employees in a similar manner and handle the errors. For editing we route our application to our /edit/id URL to begin editing the selected employee.

In our template html file, we set up the html using bootstrap for styling. We use ngFor in our <tr> tag in our table body to fill in the data for employees that our front end application retrieved from our in-memory database on initial

load (if any data is present). We finally have two buttons. An edit and delete button that once clicked call the functions implemented in our typescript class.

Employee Form Component

In our typescript file we import CommonModule and the FormsModule. The FormsModule will allow us to use two-way data binding which is something I will discuss more later. We create an empty employee property to create a new employee. We create an isEditing property and finally we create an errorMessage property.

In our constructor we inject our employee service, a router and the current activated rout to our component. In our ngOnInit function we first use the current activated route to get the web address we are currently at in our client session. If we are editing an employee, then our route will look like url/api/employee/edit/id. We subscribe to the observable that is returned from the current route which is the id as a string. If there is an id, we are in edit mode and we can set this property to true. We next call our getEmployeeById method from our employee service and subscribe to this response. We pass the id retrieved from the current route to our getEmployeeById method. The

retrieved values will populate our form and now we can begin editing fields of our form. All thislogic lives in our components ngOnInit function.

We also have our onSubmit function which is ran when a user clicks the submit or edit button on our form. We first check to see if we are in edit mode or not. We then subscribe to whatever service corresponds to the mode we are in. We finally handle any errors. Once the button is clicked, we route the session back to our home page (nav bar and table component).

For data binding in our html template, we bind data by using the [(ngModel)] attribute and setting it equal to the property that will represent our input.