Develop a simple system for managing employees in a company. Create a RESTful API in .NET Core using an ORM (EF, Dapper, etc) to connect the DB, that allows performing the following operations: Create a new employee: An employee should have a name, a salary (in USD), and a start date. Should define rules for that fields ex: name (length 100 characters), salary (with 2 decimals), start date (max value today) Get details of an employee: Create an endpoint that allows getting specific details of an employee, including the conversion currency to your local currency. \*Conversion: sould be through an API implementation with ExchangeRate-API - You need to create an account to get the Key API - Ensure proper handling of error situations, such as incorrect requests or non-existing employees. - Write unit tests to verify the functionality of at least three core features. Include tests for both success and error scenarios. Recommended: - You could use Docker to configure the entire environment with the database - Should define a Migration file - Add appropriately to a source control (Git) Evaluation Criteria: Quality in the definition of task times Use of design patterns and best practices in .NET Core development. Consideration of security in CRUD operations. Efficiency and performance in data manipulation. Proper handling of exceptions and errors.

ChatGPT

To develop a simple system for managing employees in a company using a RESTful API in .NET Core, I'll provide you with a comprehensive guide. This will include the project structure, code snippets, and an explanation of each component, such as creating, retrieving, and converting employee data while ensuring proper error handling and unit testing.

**1. Project Setup**

1. **Create a new .NET Core Web API Project:**

bash

Copy code

dotnet new webapi -n EmployeeManagement

cd EmployeeManagement

1. **Add required NuGet packages:**

bash

Copy code

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

dotnet add package Microsoft.Extensions.Http

dotnet add package Moq

dotnet add package xunit

dotnet add package FluentAssertions

**2. Define the Employee Model**

Create an Employee class in the Models directory:

csharp

Copy code

namespace EmployeeManagement.Models

{

public class Employee

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Salary { get; set; }

public DateTime StartDate { get; set; }

}

}

**3. Create the Database Context**

Create a new class EmployeeContext in the Data directory:

csharp

Copy code

using Microsoft.EntityFrameworkCore;

namespace EmployeeManagement.Data

{

public class EmployeeContext : DbContext

{

public EmployeeContext(DbContextOptions<EmployeeContext> options) : base(options) { }

public DbSet<Employee> Employees { get; set; }

}

}

**4. Configure the Database Connection**

Update appsettings.json with your database connection string:

json

Copy code

{

"ConnectionStrings": {

"EmployeeDb": "Server=localhost;Database=EmployeeDB;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

**5. Create the Employee Controller**

Create EmployeesController in the Controllers directory:

csharp

Copy code

using Microsoft.AspNetCore.Mvc;

using EmployeeManagement.Models;

using EmployeeManagement.Data;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Net.Http;

using System.Text.Json;

namespace EmployeeManagement.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class EmployeesController : ControllerBase

{

private readonly EmployeeContext \_context;

private readonly IHttpClientFactory \_httpClientFactory;

public EmployeesController(EmployeeContext context, IHttpClientFactory httpClientFactory)

{

\_context = context;

\_httpClientFactory = httpClientFactory;

}

[HttpPost]

public async Task<IActionResult> CreateEmployee(Employee employee)

{

if (string.IsNullOrEmpty(employee.Name) || employee.Name.Length > 100)

return BadRequest("Invalid name.");

if (employee.Salary < 0 || employee.Salary > 999999.99M)

return BadRequest("Invalid salary.");

if (employee.StartDate > DateTime.Now)

return BadRequest("Start date cannot be in the future.");

\_context.Employees.Add(employee);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetEmployee), new { id = employee.Id }, employee);

}

[HttpGet("{id}")]

public async Task<IActionResult> GetEmployee(int id, string currency = "USD")

{

var employee = await \_context.Employees.FindAsync(id);

if (employee == null)

return NotFound();

decimal convertedSalary = await ConvertCurrency(employee.Salary, currency);

return Ok(new { employee.Name, Salary = convertedSalary, employee.StartDate });

}

private async Task<decimal> ConvertCurrency(decimal amount, string currency)

{

var client = \_httpClientFactory.CreateClient();

var response = await client.GetStringAsync($"https://v6.exchangerate-api.com/v6/YOUR\_API\_KEY/latest/USD");

var exchangeRates = JsonSerializer.Deserialize<ExchangeRateResponse>(response);

return exchangeRates.Rates[currency] \* amount;

}

}

public class ExchangeRateResponse

{

public Dictionary<string, decimal> Rates { get; set; }

}

}

**6. Configure Dependency Injection and Database Migrations**

In Startup.cs, configure services and the database:

csharp

Copy code

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddDbContext<EmployeeContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("EmployeeDb")));

services.AddHttpClient();

}

**7. Create Migration and Update Database**

1. Create a migration:

bash

Copy code

dotnet ef migrations add InitialCreate

1. Update the database:

bash

Copy code

dotnet ef database update

**8. Implement Unit Tests**

Create a new project for testing:

bash

Copy code

dotnet new xunit -n EmployeeManagement.Tests

cd EmployeeManagement.Tests

dotnet add reference ../EmployeeManagement

Create unit tests in EmployeesControllerTests.cs:

csharp

Copy code

using EmployeeManagement.Controllers;

using EmployeeManagement.Data;

using EmployeeManagement.Models;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.DependencyInjection;

using Moq;

using System.Net.Http;

using System.Threading.Tasks;

using Xunit;

public class EmployeesControllerTests

{

private readonly EmployeeContext \_context;

public EmployeesControllerTests()

{

var options = new DbContextOptionsBuilder<EmployeeContext>()

.UseInMemoryDatabase(databaseName: "TestDatabase")

.Options;

\_context = new EmployeeContext(options);

}

[Fact]

public async Task CreateEmployee\_ReturnsCreatedResult()

{

var controller = new EmployeesController(\_context, new HttpClientFactoryMock());

var employee = new Employee { Name = "John Doe", Salary = 50000, StartDate = DateTime.Now };

var result = await controller.CreateEmployee(employee);

Assert.IsType<CreatedAtActionResult>(result);

}

[Fact]

public async Task GetEmployee\_ReturnsNotFound\_WhenEmployeeDoesNotExist()

{

var controller = new EmployeesController(\_context, new HttpClientFactoryMock());

var result = await controller.GetEmployee(999);

Assert.IsType<NotFoundResult>(result);

}

[Fact]

public async Task GetEmployee\_ReturnsOkResult\_WhenEmployeeExists()

{

var employee = new Employee { Name = "Jane Doe", Salary = 60000, StartDate = DateTime.Now };

\_context.Employees.Add(employee);

await \_context.SaveChangesAsync();

var controller = new EmployeesController(\_context, new HttpClientFactoryMock());

var result = await controller.GetEmployee(employee.Id);

Assert.IsType<OkObjectResult>(result);

}

private class HttpClientFactoryMock : IHttpClientFactory

{

public HttpClient CreateClient(string name = null)

{

return new HttpClient(new FakeHttpMessageHandler());

}

}

}

**9. Run and Test Your API**

You can run your API using:

bash

Copy code

dotnet run

You can test it using Postman or any other API testing tool by making requests to the defined endpoints.

**10. Using Docker**

To set up your application in Docker, create a Dockerfile:

dockerfile

Copy code

FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base

WORKDIR /app

EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build

WORKDIR /src

COPY ["EmployeeManagement/EmployeeManagement.csproj", "EmployeeManagement/"]

RUN dotnet restore "EmployeeManagement/EmployeeManagement.csproj"

COPY . .

WORKDIR "/src/EmployeeManagement"

RUN dotnet build "EmployeeManagement.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "EmployeeManagement.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "EmployeeManagement.dll"]

Create a docker-compose.yml:

yaml

Copy code

version: '3.4'

services:

employee-management:

image: employee-management

build:

context: .

dockerfile: EmployeeManagement/Dockerfile

ports:

- "80:80"

**11. Version Control**

1. Initialize a Git repository:

bash

Copy code

git init

git add .

git commit -m "Initial commit"

**12. Conclusion**

This guide provides a comprehensive overview of creating a RESTful API for managing employees in a .NET Core application. It covers database setup, employee management features, error handling, unit testing, Docker integration, and source control setup. Ensure you replace YOUR\_API\_KEY with your actual API key from ExchangeRate-API.

**Next Steps**

1. **Enhance Security:** Implement authentication and authorization.
2. **Documentation:** Use Swagger for API documentation.
3. **CI/CD:** Set up continuous integration and deployment pipelines.

Principio del formulario

Don't share sensitive info. Chats may be reviewed and used to train our models. [Learn more](https://help.openai.com/en/articles/5722486-how-your-data-is-used-to-improve-model-performance)

Final del formulario

ChatGPT can make mis