📝 ASP.NET Core Backend Deep Dive: Building the HR API

This guide details the steps, folder structure, core concepts, and reasoning behind building your ASP.NET Core Web API using Entity Framework Core (EF Core) and MS SQL Server.

1. Phase I: Project Foundation and Dependencies

This phase establishes the project framework and installs the necessary external libraries (NuGet packages).

1.1 Project Creation and Folder Structure

We use the standard ASP.NET Core Web API template, which provides a clean structure for RESTful APIs.

Folder/File	Purpose	Key Concept
HRDataAPI/	The root project folder.	
├ **Controllers**	Contains classes that define the **API routes** and business logic.	HTTP Endpoints (GET, POST)
├ **Data**	Contains the **Database Context**—the bridge to SQL.	EF Core Setup
├ **Models**	Defines the structure of your data (C# classes that map to SQL tables).	Code-First Approach
└ **Program.cs**	The application's entry point, handling startup, dependency injection (DI), and configuration.	Minimal Hosting Model

1.2 Installing NuGet Packages

These external libraries give the API its core functionality.

Library	Command	Why We Use It
Microsoft.EntityFrameworkCore.SqlServer	dotnet add package ...	Allows EF Core to translate C# code into T-SQL specifically for MS SQL Server.
Microsoft.EntityFrameworkCore.Design	dotnet add package ...	Provides the tools (dotnet ef) to create and manage database **migrations**.
EPPlus	dotnet add package EPPlus	A powerful library used inside the controller to read and process data from uploaded **Excel (.xlsx)** files.
Swashbuckle.AspNetCore	dotnet add package ...	Generates the **Swagger UI** for testing, debugging, and viewing API documentation in the browser.

2. Phase II: Data Modeling and Context

This is where we define the database schema purely using C# code (Code-First).

2.1 The Data Model (Models/Employee.cs)

This class maps directly to the Employees table in your SQL database.

```
// File: Models/Employee.cs
public class Employee
{
    // Primary Key: EF Core automatically sees 'Id' and sets it as the PK.
    public int Id { get; set; }

    // Nullable Reference Type: The '?' translates to a nullable VARCHAR/NVARCHAR in SQL.
    public string? FullName { get; set; }
    public string? Department { get; set; }

    public DateTime HireDate { get; set; } // Maps to datetime2 in SQL
    public decimal Salary { get; set; }    // Maps to decimal/numeric (precise currency type)
}
```

2.2 The Database Context (Data/ApplicationDbContext.cs)

The Context is the session manager that handles all communication with SQL Server.

```
// File: Data/ApplicationDbContext.cs
using Microsoft.EntityFrameworkCore;
// ...

public class ApplicationDbContext : DbContext
{
    // 1. DI Requirement: The constructor receives database options (like the connection string).
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options) {}

    // 2. DbSet: This property tells EF Core: "I need an 'Employees' table for the 'Employee'
model."
    public DbSet<Employee> Employees { get; set; }
}
```

3. Phase III: Configuration and Database Setup

This stage links your application to SQL Server and creates the database.

3.1 The Connection String (appsettings.json)

This provides the address for your local SQL Server instance.

```
// File: appsettings.json (Snippet)
"ConnectionStrings": {
    "DefaultConnection":
"Server=...\\SQLEXPRESS;Database=HRAppDB;Trusted_Connection=True;..."
}
```

3.2 Dependency Injection (DI) and CORS (Program.cs)

We use DI to make the ApplicationDbContext available wherever it's needed (like your controller).

```
// File: Program.cs (Key Code)
// 1. Register DbContext and tell it to use the connection string:
```

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// 2. Configure CORS: This allows the Angular frontend (Port 4200) to make requests
// to the API (Port 5120), bypassing browser security restrictions.
builder.Services.AddCors(options => {
    options.AddPolicy("AngularPolicy", policy =>
    {
        policy.WithOrigins("http://localhost:4200")
              .AllowAnyHeader()
              .AllowAnyMethod();
    });
});
app.UseCors("AngularPolicy"); // Enable the policy

// 3. Licensing Fix: This was necessary for EPPlus compatibility.
ExcelPackage.LicenseContext = LicenseContext.NonCommercial;
```

3.3 EF Core Migrations

Migrations are the C# developer's primary tool for managing the database schema.

Action	**Command (in HRDataAPI folder)**	**Why It's Used**
Scaffold	dotnet ef migrations add InitialCreate	Scans the models and creates the C# file containing the initial SQL instructions.
Apply	dotnet ef database update	**Crucial step:** Uses the connection string to **create the HRAppDB database (if it doesn't exist)**, and then runs the migration script to create the Employees table.

4. Phase IV: API Controller Logic

The EmployeesController handles the core application functionality.

4.1 Dependency Injection

The ApplicationDbContext is automatically provided to the controller:

```
// File: Controllers/EmployeesController.cs (Snippet)
public class EmployeesController : ControllerBase
{
    private readonly ApplicationDbContext _context;

    // DI: ASP.NET Core automatically injects the registered DbContext here.
    public EmployeesController(ApplicationDbContext context)
    {
        _context = context;
    }
    // ...
}
```

4.2 The Excel Upload Endpoint (POST /api/employees/upload)

This method is the heart of the project, handling file receipt, parsing, and saving.

Action	Code Snippet	Purpose
Receive File	public async Task<IActionResult> Upload(IFormFile file)	IFormFile is the C# type that captures the binary file uploaded from the Angular form.
Parse Data	using (var package = new ExcelPackage(stream))	Uses EPPlus to open the binary file stream for reading. The for loop iterates over each row of the sheet.
Robust Parsing	if (hireDateCell is DateTime date)	**The Fix for Excel Errors.** We used defensive code here to handle the ambiguity of Excel data (dates may come as C# DateTime objects or as raw

		double numbers). This prevents the common runtime crash.
Save to DB	_context.Employees.AddRange(newEmployees); await _context.SaveChangesAsync();	**AddRange** stages multiple records efficiently, and SaveChangesAsync executes the final batch insert into the SQL database.

4.3 The Retrieval Endpoint (GET /api/employees)

This method handles searching and ensures the data is returned in a predictable order.

```
// File: Controllers/EmployeesController.cs (Get Logic)
public async Task<ActionResult<IEnumerable<Employee>>> GetEmployees([FromQuery]
string? searchTerm)
{
    IQueryable<Employee> query = _context.Employees; // Lazy Query

    // ... (Optional filtering logic using searchTerm) ...

    // The Sorting Fix: Ensures Angular always receives records sorted by ID.
    // We set Descending (highest ID first) to force a change and ensure the rule applies.
    return await query.OrderByDescending(e => e.Id).ToListAsync();
}
```