Entity Framework Core (EF Core) Integration with ASP.NET Core Web API

Entity Framework Core (EF Core) is an Object-Relational Mapping (ORM) framework for .NET that simplifies database interactions by mapping .NET objects to database tables. When integrated with ASP.NET Core Web API, it provides a powerful way to build data-driven RESTful services. This material covers setting up EF Core, configuring an in-memory database, and implementing CRUD operations in a Web API.

1. Overview of EF Core and ASP.NET Core Web API

What is EF Core?

- EF Core is a lightweight, extensible ORM for .NET.
- It allows developers to work with data using C# objects instead of raw SQL.
- Supports multiple database providers (e.g., SQL Server, SQLite, In-Memory).

Why Use EF Core with Web API?

- Simplifies data access and manipulation.
- Provides features like change tracking, migrations, and LINQ queries.
- Integrates seamlessly with ASP.NET Core's dependency injection.

In-Memory Database

- For this guide, we'll use EF Core's **In-Memory Database** provider. It's ideal for learning and testing without requiring an external database.

2. Setting Up the Project

Prerequisites

- .NET SDK (e.g., .NET 8 as of March 2025).
- IDE (Visual Studio, VS Code, or similar).
- Basic understanding of ASP.NET Core Web API (from the previous material).

Create a New Web API Project

```
dotnet new webapi -o EfCoreApi
cd EfCoreApi
```

Add EF Core Packages

Install the necessary NuGet packages via the .NET CLI:

```
dotnet add package Microsoft.EntityFrameworkCore.InMemory
dotnet add package Microsoft.EntityFrameworkCore
```

- `Microsoft.EntityFrameworkCore.InMemory`: Provides the in-memory database provider.
- `Microsoft.EntityFrameworkCore`: Core EF functionality.

Project Structure

After setup, we'll add:

- **Models**: Define the data structure.
- **Data**: EF Core context for database access.
- **Controllers**: API endpoints.

---

3. Configuring EF Core

Define the Model

Create a `Product` model similar to the previous example.

```
// Models/Product.cs
namespace EfCoreApi.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

Create the DbContext

The `DbContext` class represents the database and provides access to data.

```
// Data/AppDbContext.cs
using EfCoreApi.Models;
using Microsoft.EntityFrameworkCore;

namespace EfCoreApi.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options)
        {
        }
        public DbSet<Product> Products { get; set; }
    }
}
```

Configure EF Core in Program.cs

Register the `DbContext` with the in-memory database provider.

```
// Program.cs
using EfCoreApi.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseInMemoryDatabase("InMemoryDb")); // Configure in-memory database

var app = builder.Build();

// Configure the HTTP request pipeline
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

Seed Initial Data (Optional)

To mimic the in-memory list from the previous example, seed data at startup.

```
// Program.cs (continued)
using EfCoreApi.Data;
using EfCoreApi.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseInMemoryDatabase("InMemoryDb"));

var app = builder.Build();

// Seed data
using (var scope = app.Services.CreateScope())
```

```
{
    var context = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    context.Products.AddRange(
        new Product { Id = 1, Name = "Laptop", Price = 999.99m },
        new Product { Id = 2, Name = "Mouse", Price = 19.99m }
    );
    context.SaveChanges();
}

// Configure the HTTP request pipeline
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

4. Implementing CRUD Operations with EF Core

Create the API Controller

Replace the previous `ProductService` with direct EF Core operations in the controller.

```
// Controllers/ProductsController.cs
using EfCoreApi.Data;
using EfCoreApi.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace EfCoreApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        private readonly AppDbContext _context;

        public ProductsController(AppDbContext context)
        {
            _context = context;
        }

        // GET: api/products
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Product>>> GetAll()
        {
            return await _context.Products.ToListAsync();
        }

        // GET: api/products/1
        [HttpGet("{id}")]
        public async Task<ActionResult<Product>> GetById(int id)
        {
            var product = await _context.Products.FindAsync(id);
            if (product == null) return NotFound();
            return product;
        }

        // POST: api/products
        [HttpPost]
        public async Task<ActionResult<Product>> Create([FromBody] Product
product)
        {
            _context.Products.Add(product);
            await _context.SaveChangesAsync();
            return CreatedAtAction(nameof(GetById), new { id = product.Id },
product);
        }

        // PUT: api/products/1
        [HttpPut("{id}")]
        public async Task<IActionResult> Update(int id, [FromBody] Product
product)
        {
            if (id != product.Id) return BadRequest();
            _context.Entry(product).State = EntityState.Modified;
            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!ProductExists(id)) return NotFound();
                throw;
            }
            return NoContent();
        }

        // DELETE: api/products/1
        [HttpDelete("{id}")]
        public async Task<IActionResult> Delete(int id)
        {
            var product = await _context.Products.FindAsync(id);
            if (product == null) return NotFound();
            _context.Products.Remove(product);
            await _context.SaveChangesAsync();
            return NoContent();
        }

        private bool ProductExists(int id)
        {
            return _context.Products.Any(p => p.Id == id);
        }
    }
}
```

Key Changes from In-Memory List

- **Async/Await**: EF Core operations are asynchronous, improving scalability.
- **DbContext**: Replaces the service class, interacting directly with the database.
- **Change Tracking**: EF Core tracks changes to entities and persists them with `SaveChangesAsync()`.

---

5. Adding Data Validation

Update the Model with Annotations

Add validation rules using `System.ComponentModel.DataAnnotations`.

```
// Models/Product.cs
using System.ComponentModel.DataAnnotations;

namespace EfCoreApi.Models
{
    public class Product
    {
        public int Id { get; set; }

        [Required(ErrorMessage = "Name is required")]
        [StringLength(50, MinimumLength = 2, ErrorMessage = "Name must be between 2 and 50 characters")]
        public string Name { get; set; }

        [Range(0.01, 10000.00, ErrorMessage = "Price must be between 0.01 and 10000.00")]
        public decimal Price { get; set; }
    }
}
```

Validate in the Controller

Leverage `ModelState` for validation.

Update `Create`

```
[HttpPost]
public async Task<ActionResult<Product>> Create([FromBody] Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    _context.Products.Add(product);
    await _context.SaveChangesAsync();
    return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

Update `Update`

```
[HttpPut("{id}")]
public async Task<IActionResult> Update(int id, [FromBody] Product product)
{
    if (id != product.Id) return BadRequest();
    if (!ModelState.IsValid) return BadRequest(ModelState);

    _context.Entry(product).State = EntityState.Modified;
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!ProductExists(id)) return NotFound();
        throw;
    }
    return NoContent();
}
```

---

6. Running and Testing the API

Run the Project

```
dotnet run
```

- API runs at `https://localhost:5001` (or similar).

Test Endpoints

Use Postman or cURL:

1. **GET All**: `GET https://localhost:5001/api/products`
   - Response: JSON list of seeded products.
2. **POST**: `POST https://localhost:5001/api/products`
   - Body: `{"name": "Keyboard", "price": 49.99}`
   - Response: 201 Created.
3. **GET by ID**: `GET https://localhost:5001/api/products/3`
4. **PUT**: `PUT https://localhost:5001/api/products/3`
   - Body: `{"id": 3, "name": "Updated Keyboard", "price": 59.99}`

5. **DELETE**: `DELETE https://localhost:5001/api/products/3`

Validation Test

- **Invalid POST** (e.g., missing `Name`):

  ```
  {
      "price": 10.99
  }
  ```

  Response (HTTP 400):

  ```
  {
      "errors": {
          "Name": ["Name is required"]
      }
  }
  ```

---

7. Practical Application Example

This API now uses EF Core to manage a product catalog persisted in memory. Key differences from the in-memory list approach:

- Data is managed by EF Core's change tracking.
- Operations are asynchronous, aligning with real-world database scenarios.
- Validation ensures data integrity before persistence.

Summary Table

Topic	Description	Key Feature
Project Setup	Add EF Core to Web API	`UseInMemoryDatabase`
EF Core Configuration	DbContext and model setup	`AppDbContext`, `DbSet`
CRUD Operations	Use EF Core for data access	Async methods, LINQ
Data Validation	Enforce rules on model	Annotations, `ModelState`

Exercises

1. Add a `Category` property to `Product` and filter products by category with a new endpoint (`GET api/products/category/{category}`).
2. Implement a custom validation rule (e.g., `Name` must not contain numbers) using a custom attribute.
3. Seed additional data and create an endpoint to reset the database to its initial state.