Part 1: Setting Up the Project and CRUD for Products

1. Create a New .NET Core Web API Project

- Open your terminal or IDE (like Visual Studio or VS Code) and create a new project.

```
dotnet new webapi -n ECommerceAPI
cd ECommerceAPI
```

2. Install Required NuGet Packages

Install `EntityFrameworkCore` packages to work with the database.

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

3. Set Up Database Context

- Create a folder named `Data`.
- Inside the `Data` folder, create a file `ECommerceDbContext.cs`:

```
using Microsoft.EntityFrameworkCore;

namespace ECommerceAPI.Data
{
	public class ECommerceDbContext : DbContext
	{
	public ECommerceDbContext(DbContextOptions<ECommerceDbContext> options) :
base(options) { }

	public DbSet<Product> Products { get; set; }
	}
}
```

4. Create the Product Entity

- Create a folder named `Models`.
- Inside the `Models` folder, create a file `Product.cs`:

```
namespace ECommerceAPI.Models
{
	public class Product
	{
	public int Id { get; set; }
	public string Name { get; set; }
	public string Description { get; set; }
	public decimal Price { get; set; }
	public int Stock { get; set; }
	}
}
```

5. Configure the Database

- Open `appsettings.json` and add a connection string:

```
"ConnectionStrings": {
  "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=ECommerceDB;Trusted_Connection=True;MultipleA
ctiveResultSets=true"
}
```

Register the database context in `Program.cs`:

```
using ECommerceAPI.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<ECommerceDbContext>(options =>
	options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
	app.UseSwagger();
```

```
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

6. Create the Initial Migration

Run the following commands to create and apply the initial migration:

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

6.1

1. Install the `dotnet-ef` Tool Globally

Run the following command to install the `dotnet-ef` CLI tool globally:

```
dotnet tool install --global dotnet-ef
```

Run a New SQL Server Container with a Different Name and Port

Use a unique name for the container and map it to a different port (e.g., 1434):

```
docker run -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=AnotherStrong@Password" -p 1434:1433 --name sqlserver2 -d mcr.microsoft.com/mssql/server:2019-latest
```

Container Name: `sqlserver2`
Host Port: `1434` (mapped to the container's default SQL Server port `1433`)
Password: `AnotherStrong@Password`

Update the .NET Core App for the New Instance

1. Modify the connection string in `appsettings.json` to point to the new port and database:

```
"ConnectionStrings": {
  "DefaultConnection": "Server=localhost,1434;Database=ECommerceDB2;User
Id=SA;Password=AnotherStrong@Password;"
},
```

- **Server**: `localhost,1434` (the new port).
- **Database**: `ECommerceDB2` (the new database name).
- **Password**: `AnotherStrong@Password`.

The rest of the configuration in `Program.cs` remains the same, as shown earlier.

Verify the New Database

You can use any SQL client like SQL Server Management Studio (SSMS) or Azure Data Studio to connect to `localhost,1434` and verify that `ECommerceDB2` exists.

Apply Migrations to the New Database

Run the following commands to set up the new database schema:

1. **Add a new migration** (if not done yet):

```
dotnet ef migrations add InitialCreate
```

Apply the migration to the new database:

```
dotnet ef database update
```

7. Create the Products Controller

- Create a folder named `Controllers`.
- Inside `Controllers`, create a file `ProductsController.cs`:

```
using ECommerceAPI.Data;
using ECommerceAPI.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace ECommerceAPI.Controllers
```

```
{
        [Route("api/[controller]")]
        [ApiController]
        public class ProductsController : ControllerBase
        {
        private readonly ECommerceDbContext _context;

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

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

        // GET: api/Products/5
        [HttpGet("{id}")]
        public async Task<ActionResult<Product>> GetProduct(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>> PostProduct(Product product)
        {
        _context.Products.Add(product);
        await _context.SaveChangesAsync();

        return CreatedAtAction(nameof(GetProduct), new { id = product.Id }, product);
        }

        // PUT: api/Products/5
```

```
[HttpPut("{id}")]
public async Task<IActionResult> PutProduct(int id, 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();
        }
        else
        {
        throw;
        }
}

return NoContent();
}

// DELETE: api/Products/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteProduct(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(e => e.Id == id);
        }
        }
}
```

8. Test the API

- Run the application:

dotnet run

Use a tool like Postman, Swagger, or curl to test the CRUD endpoints:

- `GET /api/Products`
- `GET /api/Products/{id}`
- `POST /api/Products`
- `PUT /api/Products/{id}`
- `DELETE /api/Products/{id}`