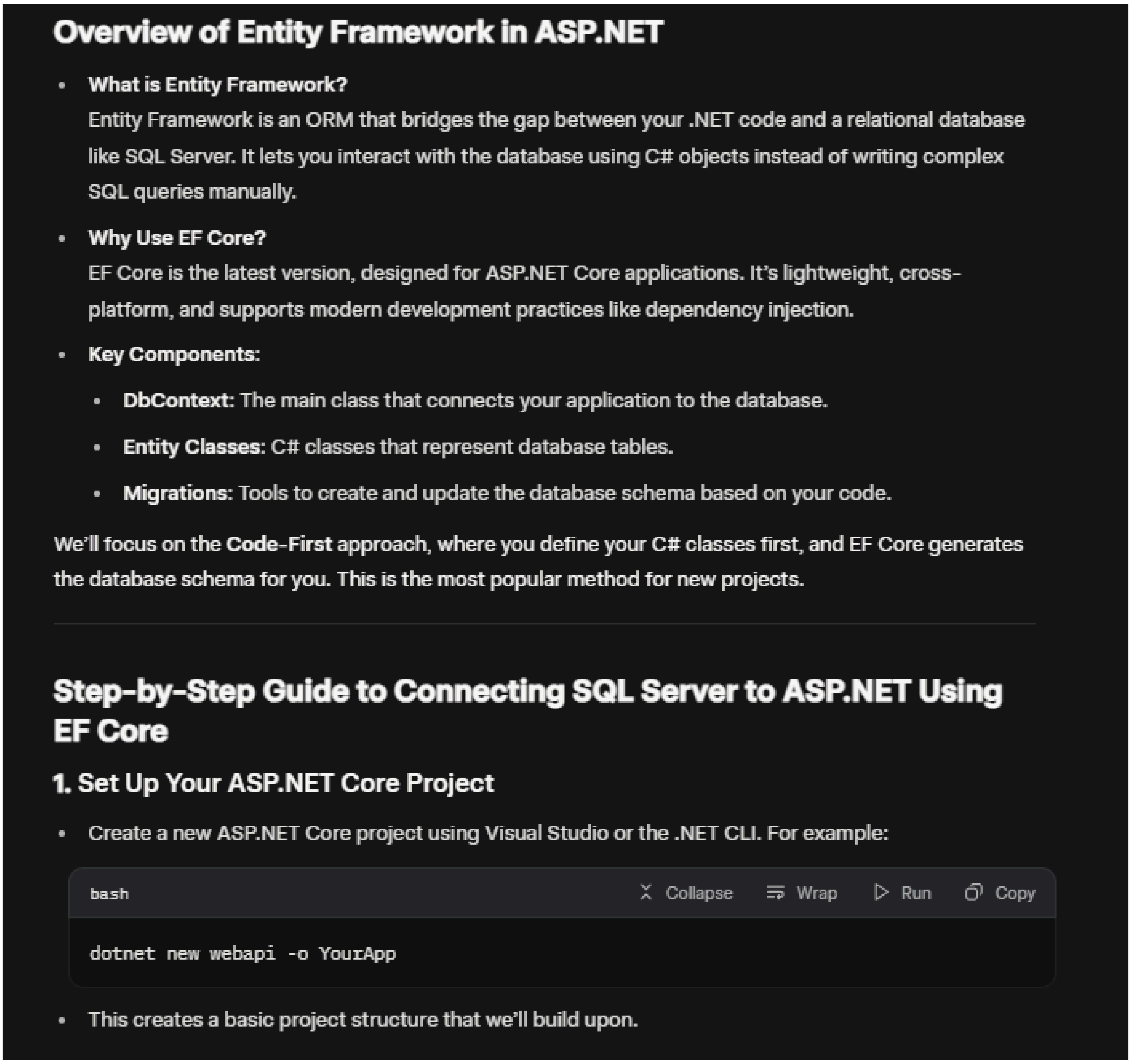


Overview of Entity Framework in ASP.NET

- **What is Entity Framework?**
  Entity Framework is an ORM that bridges the gap between your .NET code and a relational database like SQL Server. It lets you interact with the database using C# objects instead of writing complex SQL queries manually.
- **Why Use EF Core?**
  EF Core is the latest version, designed for ASP.NET Core applications. It's lightweight, cross-platform, and supports modern development practices like dependency injection.
- **Key Components:**
  - **DbContext**: The main class that connects your application to the database.
  - **Entity Classes**: C# classes that represent database tables.
  - **Migrations**: Tools to create and update the database schema based on your code.

We'll focus on the **Code-First** approach, where you define your C# classes first, and EF Core generates the database schema for you. This is the most popular method for new projects.

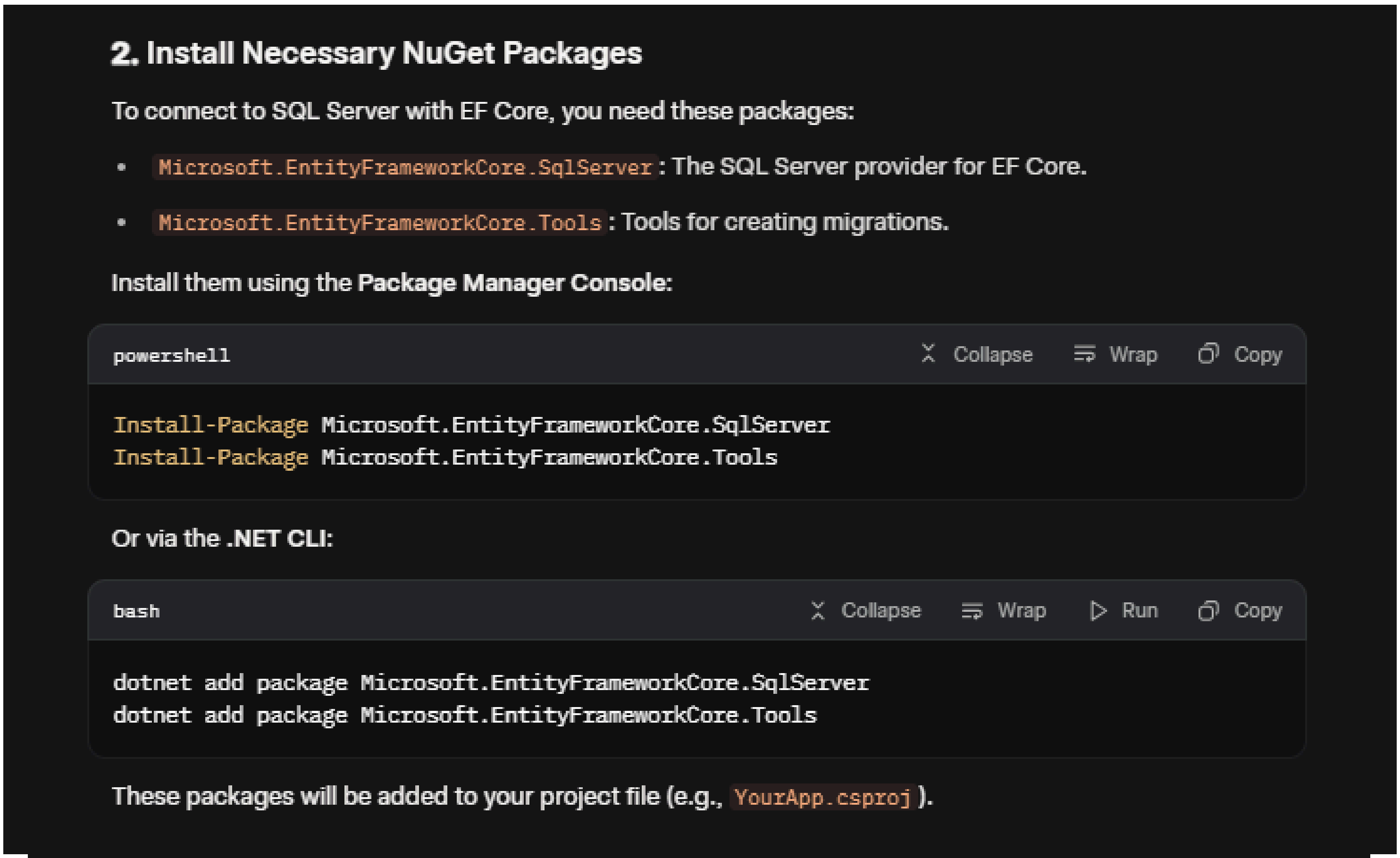
Step-by-Step Guide to Connecting SQL Server to ASP.NET Using EF Core

1. Set Up Your ASP.NET Core Project

- Create a new ASP.NET Core project using Visual Studio or the .NET CLI. For example:

```bash
dotnet new webapi -o YourApp
```

- This creates a basic project structure that we'll build upon.



2. Install Necessary NuGet Packages

To connect to SQL Server with EF Core, you need these packages:

- `Microsoft.EntityFrameworkCore.SqlServer`: The SQL Server provider for EF Core.
- `Microsoft.EntityFrameworkCore.Tools`: Tools for creating migrations.

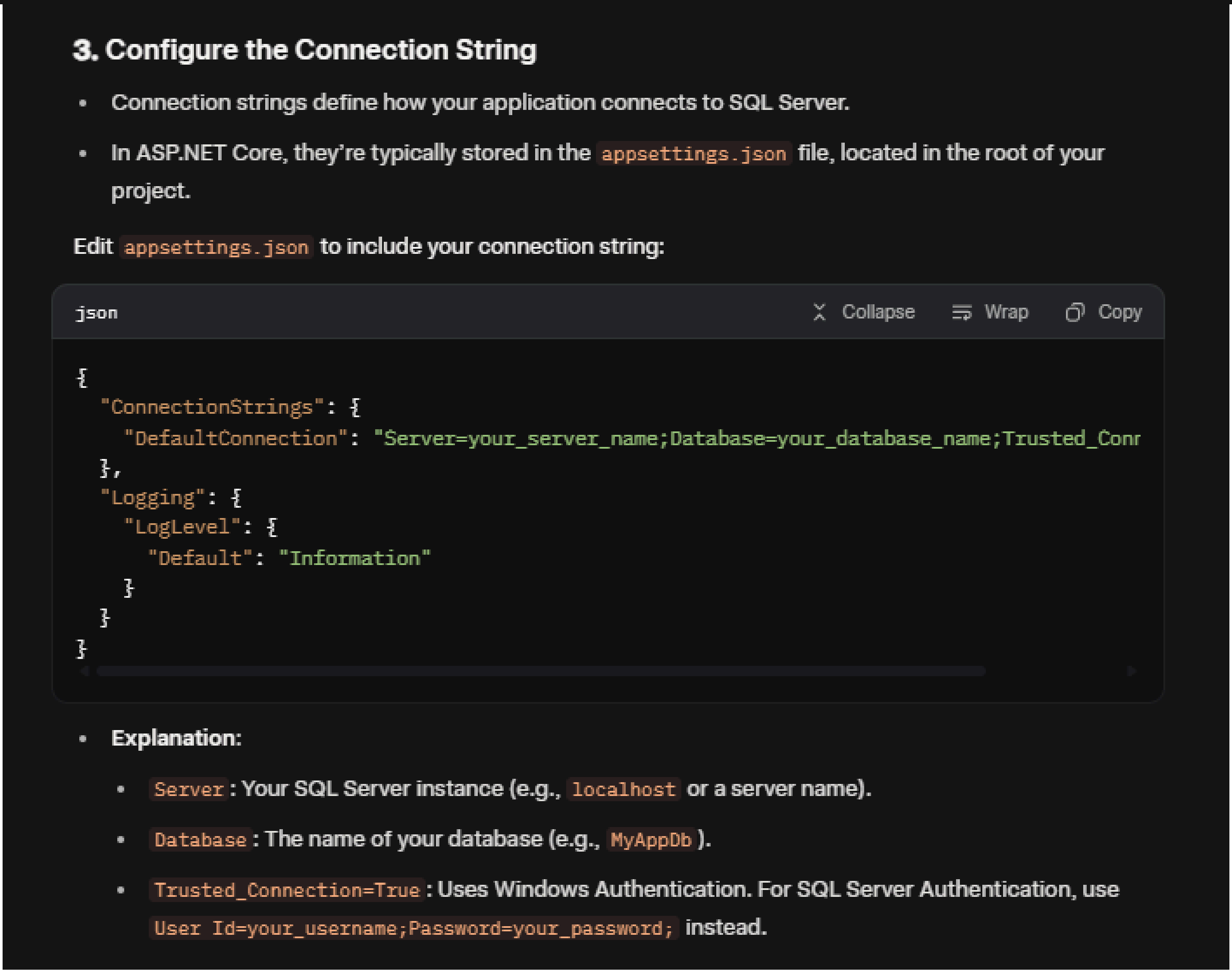
Install them using the **Package Manager Console:**

```powershell
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Tools
```

Or via the **.NET CLI:**

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

These packages will be added to your project file (e.g., `YourApp.csproj`).



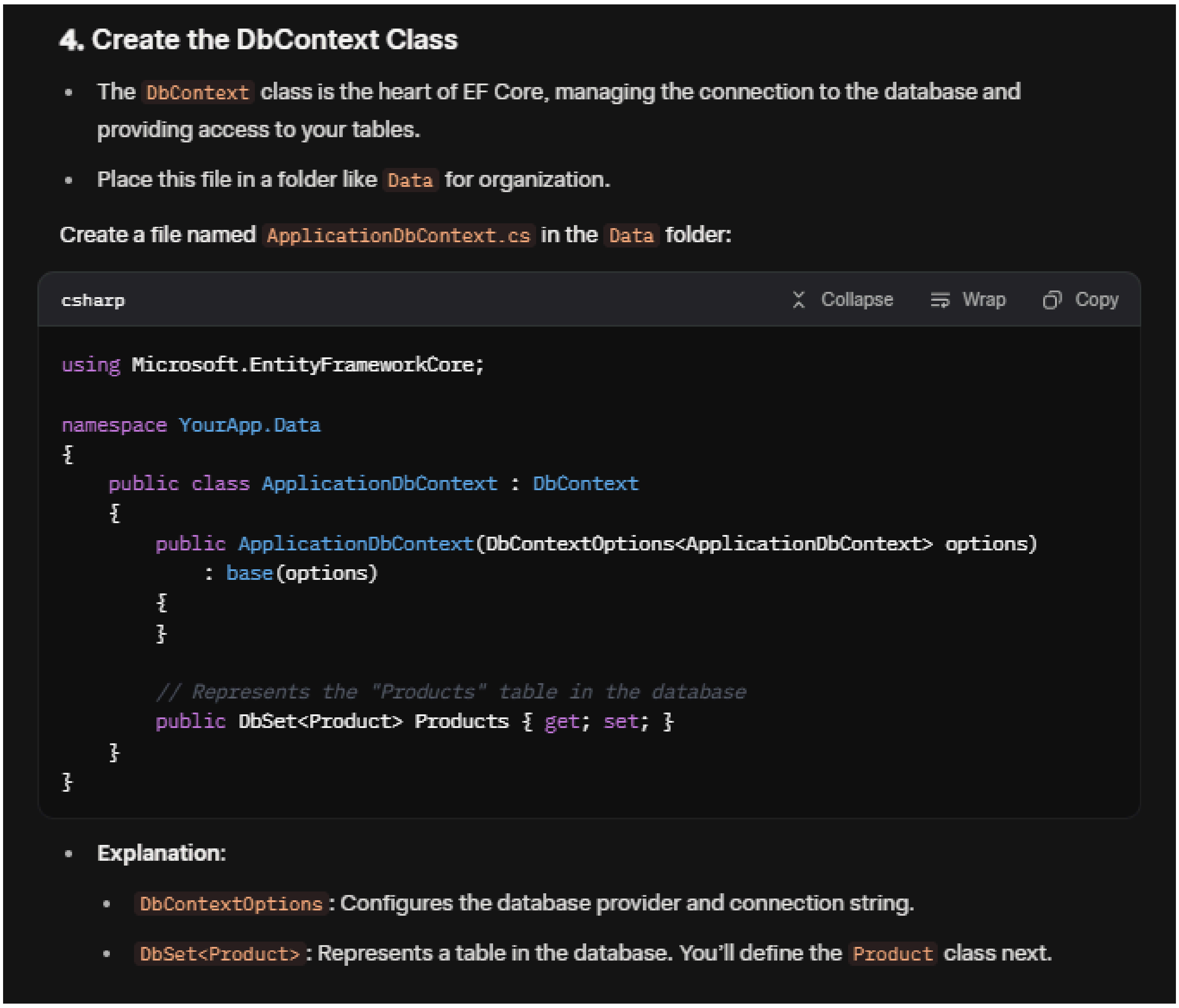
3. Configure the Connection String

- Connection strings define how your application connects to SQL Server.
- In ASP.NET Core, they're typically stored in the `appsettings.json` file, located in the root of your project.

Edit `appsettings.json` to include your connection string:

```json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=your_server_name;Database=your_database_name;Trusted_Conr
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

- **Explanation:**
  - `Server`: Your SQL Server instance (e.g., `localhost` or a server name).
  - `Database`: The name of your database (e.g., `MyAppDb`).
  - `Trusted_Connection=True`: Uses Windows Authentication. For SQL Server Authentication, use `User Id=your_username;Password=your_password;` instead.



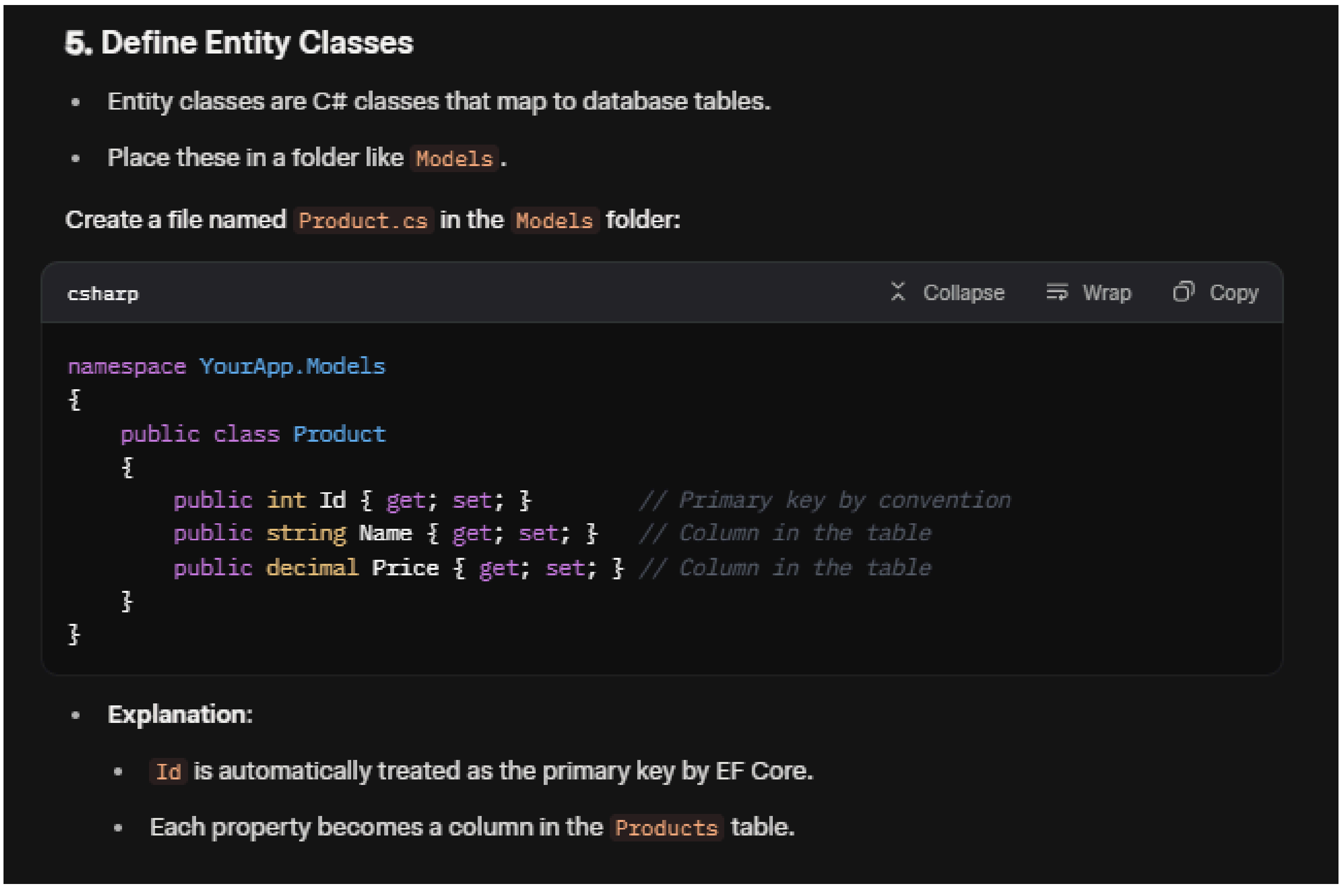
4. Create the DbContext Class

- The `DbContext` class is the heart of EF Core, managing the connection to the database and providing access to your tables.
- Place this file in a folder like `Data` for organization.

Create a file named `ApplicationDbContext.cs` in the `Data` folder:

```csharp
using Microsoft.EntityFrameworkCore;

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

        // Represents the "Products" table in the database
        public DbSet<Product> Products { get; set; }
    }
}
```

- **Explanation:**
  - `DbContextOptions`: Configures the database provider and connection string.
  - `DbSet<Product>`: Represents a table in the database. You'll define the `Product` class next.



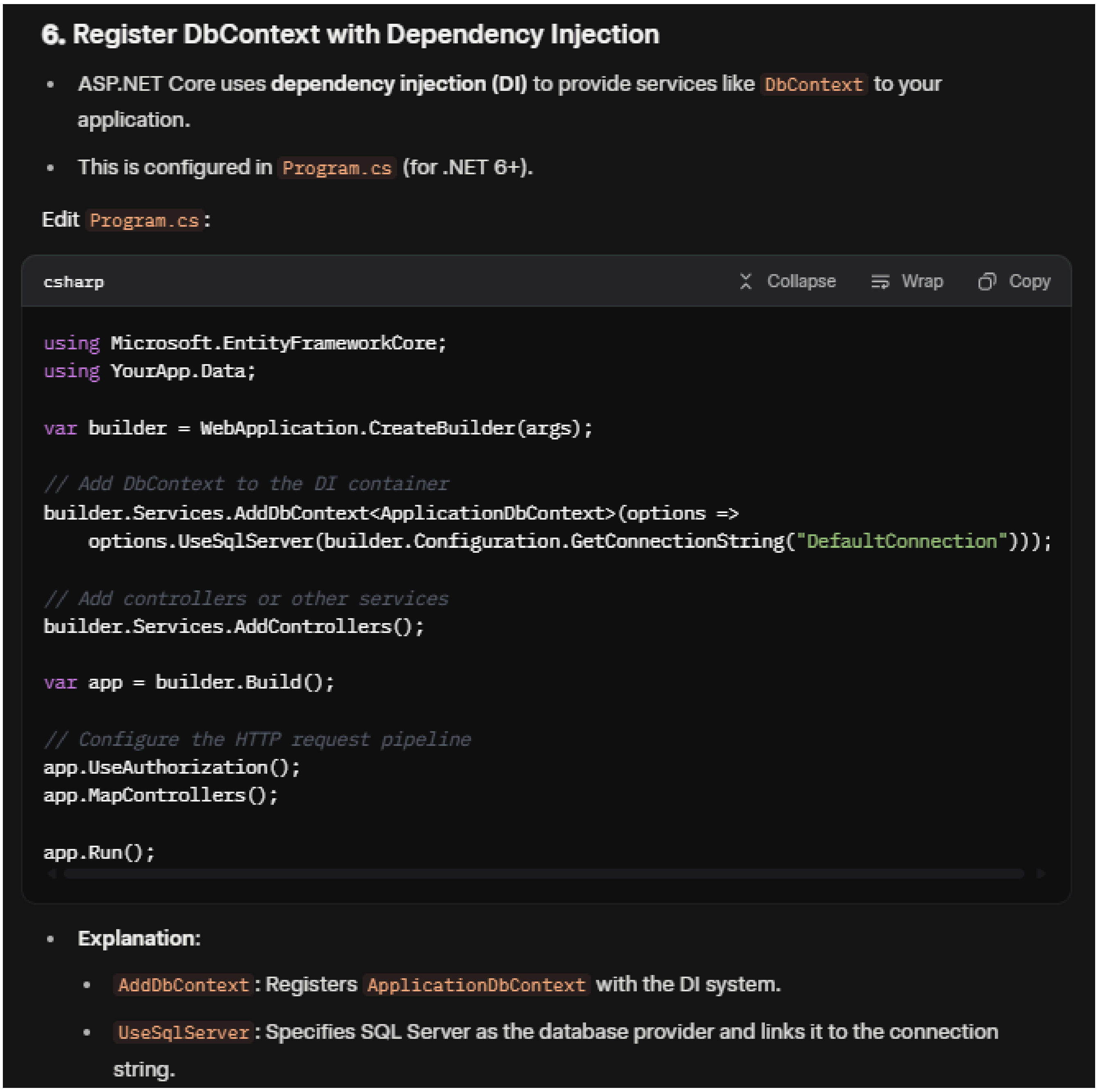
5. Define Entity Classes

- Entity classes are C# classes that map to database tables.
- Place these in a folder like `Models`.

Create a file named `Product.cs` in the `Models` folder:

```csharp
namespace YourApp.Models
{
    public class Product
    {
        public int Id { get; set; }         // Primary key by convention
        public string Name { get; set; }    // Column in the table
        public decimal Price { get; set; } // Column in the table
    }
}
```

- **Explanation:**
  - `Id` is automatically treated as the primary key by EF Core.
  - Each property becomes a column in the `Products` table.



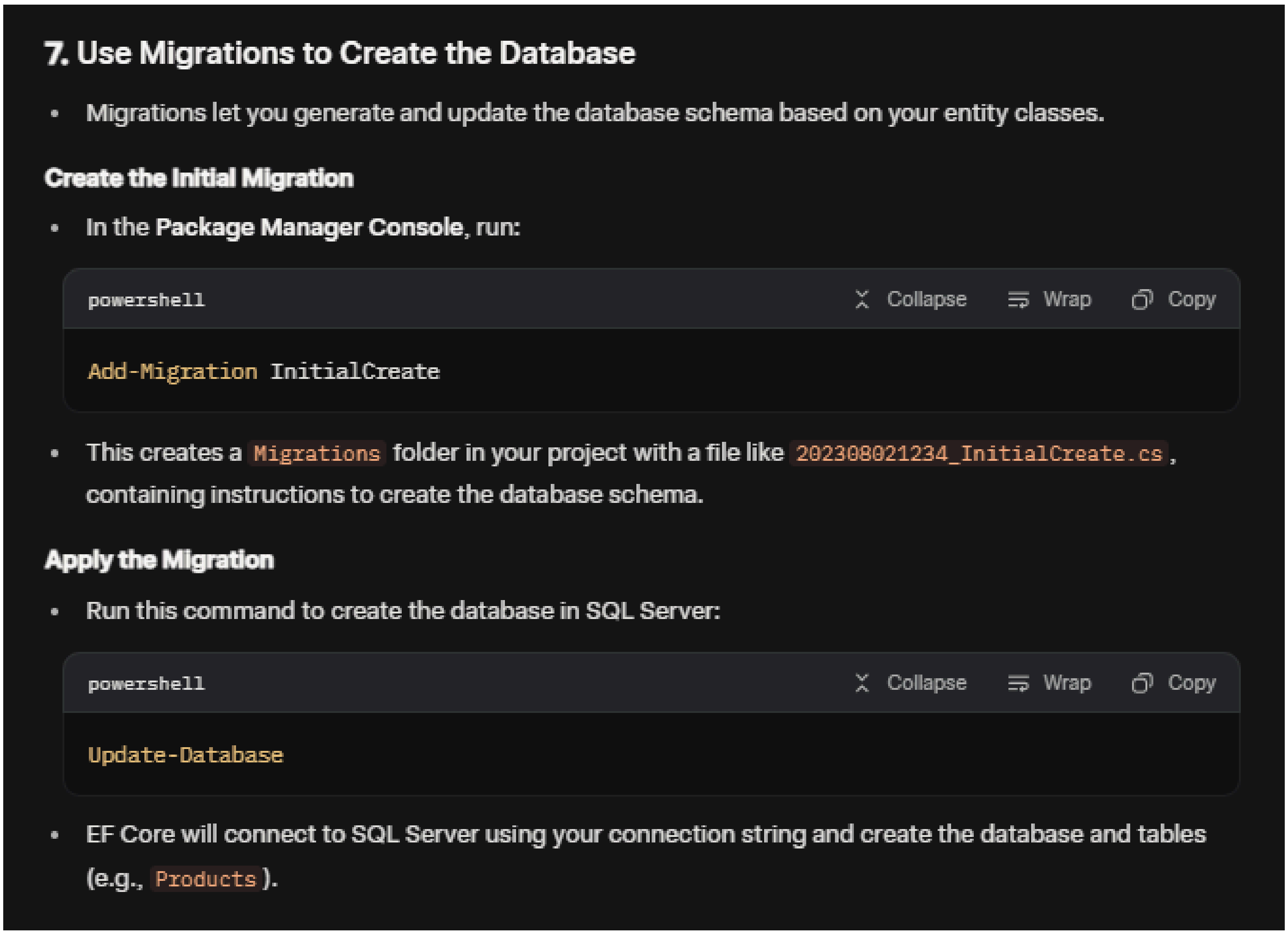
6. Register DbContext with Dependency Injection

- ASP.NET Core uses **dependency injection (DI)** to provide services like `DbContext` to your application.
- This is configured in `Program.cs` (for .NET 6+).

Edit `Program.cs`:

```csharp
using Microsoft.EntityFrameworkCore;
using YourApp.Data;

var builder = WebApplication.CreateBuilder(args);

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

// Add controllers or other services
builder.Services.AddControllers();

var app = builder.Build();

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

app.Run();
```

- **Explanation:**
  - `AddDbContext`: Registers `ApplicationDbContext` with the DI system.
  - `UseSqlServer`: Specifies SQL Server as the database provider and links it to the connection string.



7. Use Migrations to Create the Database

- Migrations let you generate and update the database schema based on your entity classes.

Create the Initial Migration

- In the **Package Manager Console**, run:

```powershell
Add-Migration InitialCreate
```

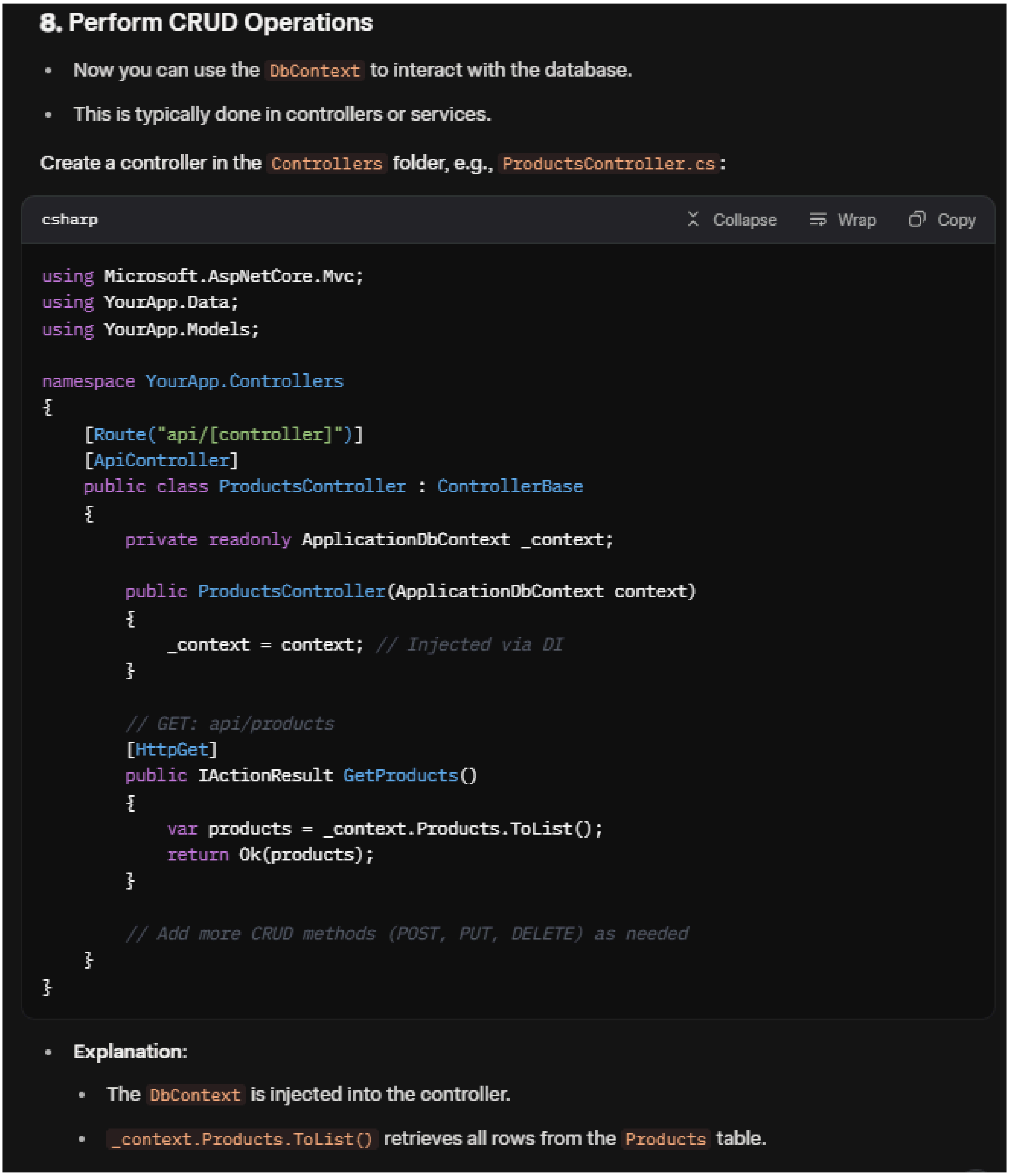
- This creates a `Migrations` folder in your project with a file like `202308021234_InitialCreate.cs`, containing instructions to create the database schema.

Apply the Migration

- Run this command to create the database in SQL Server:

```powershell
Update-Database
```

- EF Core will connect to SQL Server using your connection string and create the database and tables (e.g., `Products`).



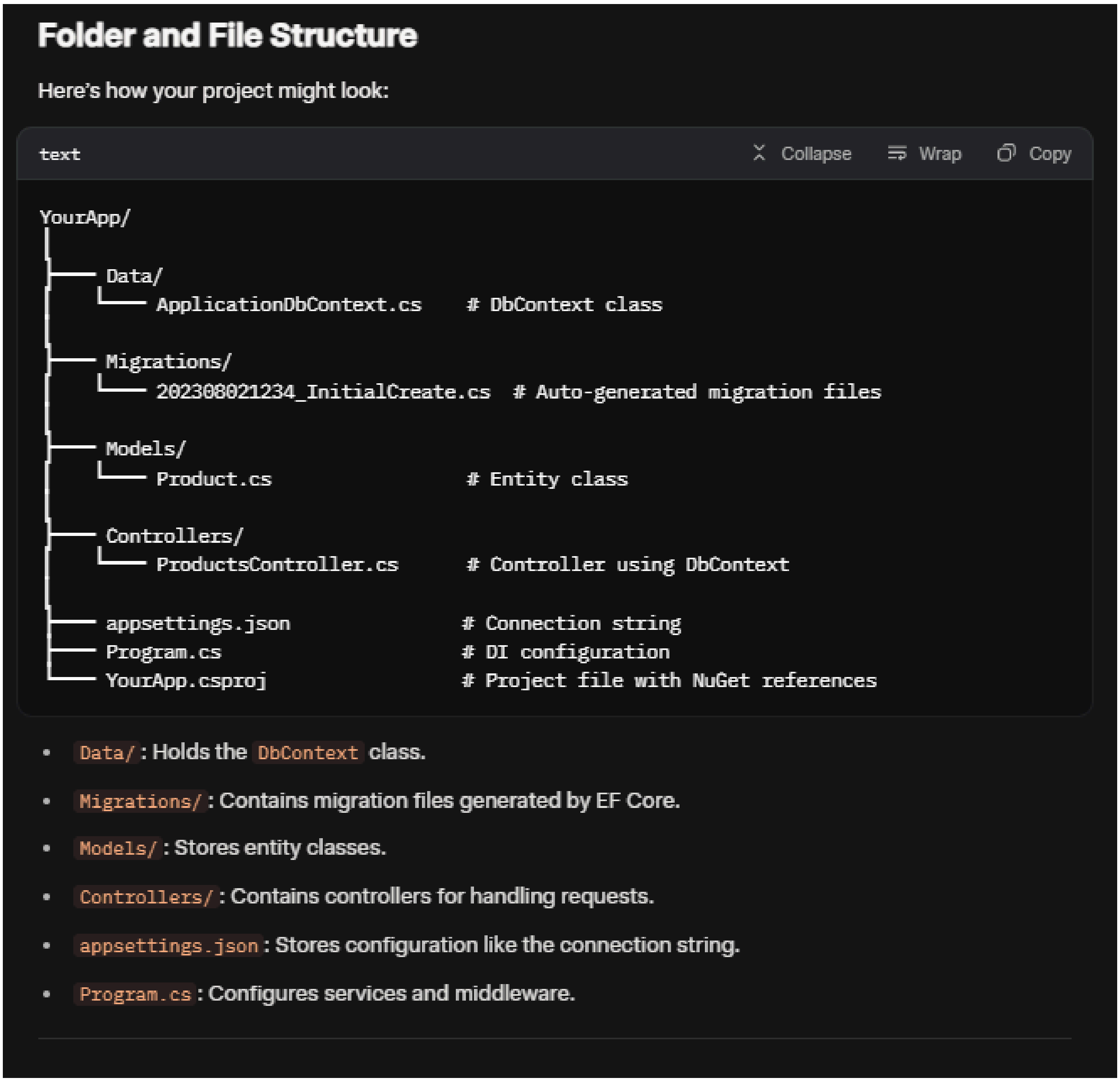
8. Perform CRUD Operations

- Now you can use the `DbContext` to interact with the database.
- This is typically done in controllers or services.

Create a controller in the `Controllers` folder, e.g., `ProductsController.cs`:

```csharp
using Microsoft.AspNetCore.Mvc;
using YourApp.Data;
using YourApp.Models;

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

        public ProductsController(ApplicationDbContext context)
        {
            _context = context; // Injected via DI
        }

        // GET: api/products
        [HttpGet]
        public IActionResult GetProducts()
        {
            var products = _context.Products.ToList();
            return Ok(products);
        }

        // Add more CRUD methods (POST, PUT, DELETE) as needed
    }
}
```

- **Explanation:**
  - The `DbContext` is injected into the controller.
  - `_context.Products.ToList()` retrieves all rows from the `Products` table.

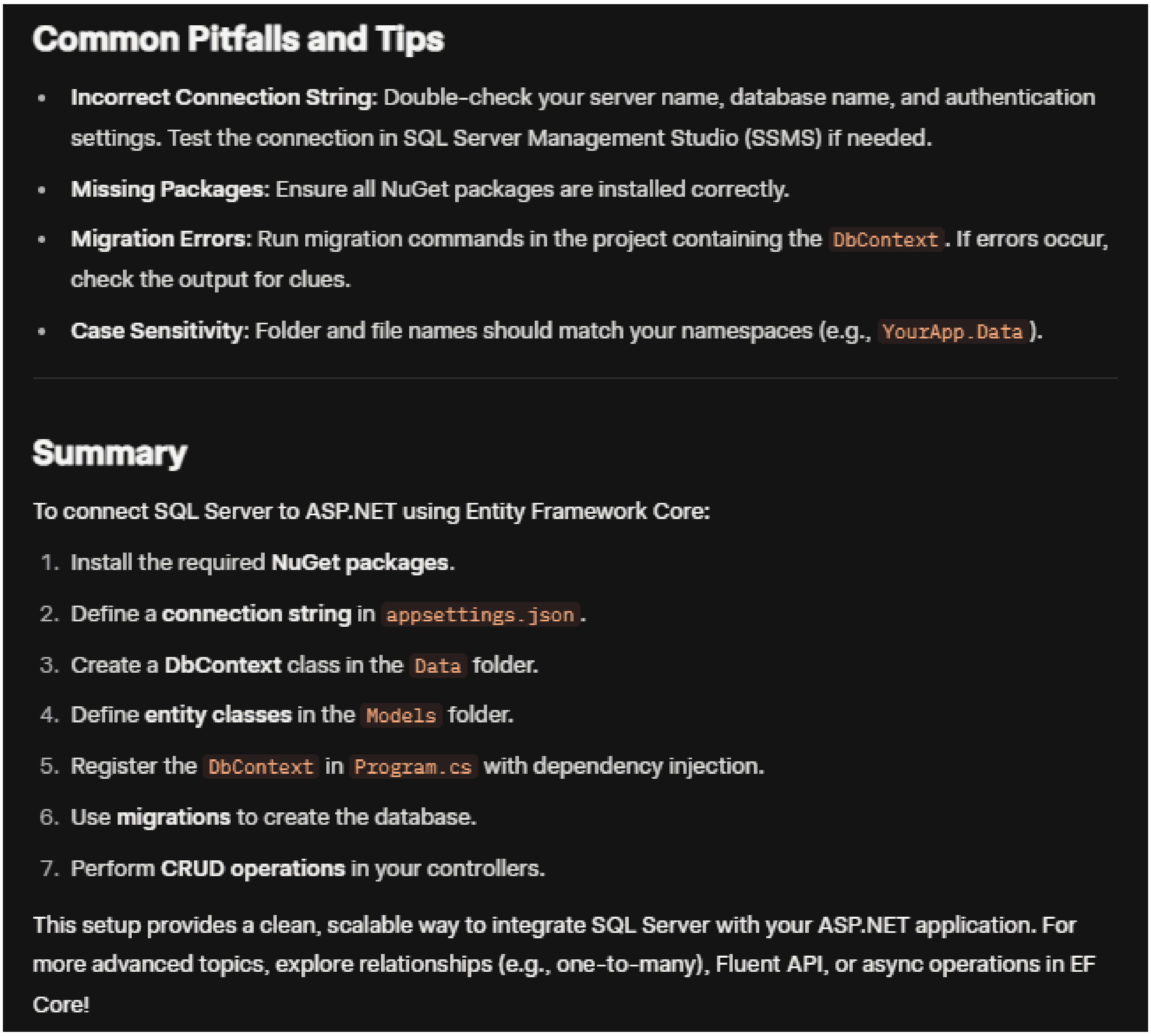


Folder and File Structure

Here's how your project might look:

```text
YourApp/
│
├── Data/
│   └── ApplicationDbContext.cs    # DbContext class
│
├── Migrations/
│   └── 202308021234_InitialCreate.cs  # Auto-generated migration files
│
├── Models/
│   └── Product.cs                 # Entity class
│
├── Controllers/
│   └── ProductsController.cs      # Controller using DbContext
│
├── appsettings.json               # Connection string
├── Program.cs                     # DI configuration
└── YourApp.csproj                 # Project file with NuGet references
```

- `Data/` : Holds the `DbContext` class.
- `Migrations/` : Contains migration files generated by EF Core.
- `Models/` : Stores entity classes.
- `Controllers/` : Contains controllers for handling requests.
- `appsettings.json` : Stores configuration like the connection string.
- `Program.cs` : Configures services and middleware.



Common Pitfalls and Tips

- **Incorrect Connection String:** Double-check your server name, database name, and authentication settings. Test the connection in SQL Server Management Studio (SSMS) if needed.
- **Missing Packages:** Ensure all NuGet packages are installed correctly.
- **Migration Errors:** Run migration commands in the project containing the `DbContext`. If errors occur, check the output for clues.
- **Case Sensitivity:** Folder and file names should match your namespaces (e.g., `YourApp.Data`).

Summary

To connect SQL Server to ASP.NET using Entity Framework Core:

1. Install the required **NuGet packages**.
2. Define a **connection string** in `appsettings.json`.
3. Create a **DbContext** class in the `Data` folder.
4. Define **entity classes** in the `Models` folder.
5. Register the `DbContext` in `Program.cs` with dependency injection.
6. Use **migrations** to create the database.
7. Perform **CRUD operations** in your controllers.

This setup provides a clean, scalable way to integrate SQL Server with your ASP.NET application. For more advanced topics, explore relationships (e.g., one-to-many), Fluent API, or async operations in EF Core!