⚡ Step 1: Install .NET 8 SDK (LTS)

1. Go to the official download page:
   👉 Download .NET 8 SDK

2. Choose the **SDK installer** (not just the runtime).

   - SDK = lets you **develop & run** apps.
   - Runtime = lets you **run only** apps (not what we need).

3. Install it just like any other program (click **Next** → **Next** → **Finish**).

✅ Verify Installation

After installation, open **Command Prompt** or **PowerShell** and run:

dotnet --version

If it shows something like:

8.0.xxx

🎉 Congrats! You have .NET 8 installed.

🗂️ ASP.NET Core Web API Project Structure (Quick Guide)

When you open the project, you'll see some folders/files. Let's connect them to real life so it's easier to remember 👇

📦 bin/ (Finished Products)

- **Real life:** Think of it like a warehouse where your finished goods are stored.
- **Tech:** Contains compiled DLLs (output of your app). Appears after you **build/run** the project.

⚙️ obj/ (Factory Machines)

- **Real life:** Machines/tools that help make the product but not part of the final delivery.
- **Tech:** Temporary build files (metadata, caches). Used internally by the compiler.

🏠 Properties/ (House Blueprint)

- **Real life:** Like instructions on how to open your house (doors, windows).
- **Tech:** Contains launchSettings.json, which controls **how the project runs** (port, HTTPS, etc.).

📝 appsettings.json (Company Handbook)

- **Real life:** Rules everyone in the company follows.
- **Tech:** Global configuration file (logging, DB connections, custom settings).

📝 appsettings.Development.json (Manager's Notes)

- **Real life:** Extra rules your manager gives you only while in training.
- **Tech:** Overrides appsettings.json when running in **Development mode**.

💻 Program.cs (Switchboard)

- **Real life:** The main switch that powers everything ON.
- **Tech:** Entry point → registers services, sets up routing, middleware, Swagger, etc.

📄 StudentWebApi.csproj (Project Contract)

- **Real life:** Contract listing what tools/materials your company uses.
- **Tech:** Project file → defines SDK, dependencies (NuGet), and build settings.

🌐 StudentWebApi.http (Test Orders)

- **Real life:** Like sending a sample order to check if the system works.
- **Tech:** Lets you test API endpoints directly in VS/VS Code (like Postman).

⚡ Important Setup: Change Port

1. Open your project folder and go to:
   Properties → launchSettings.json

Once you open it, you'll notice there are **two entries** under **"applicationUrl"** → one for **HTTP** and one for **HTTPS**.

For now, kindly **change only the HTTP port** to **9090** so we all run the same config:

"applicationUrl": "https://localhost:7243;http://localhost:9090"

❓ Why only HTTP?

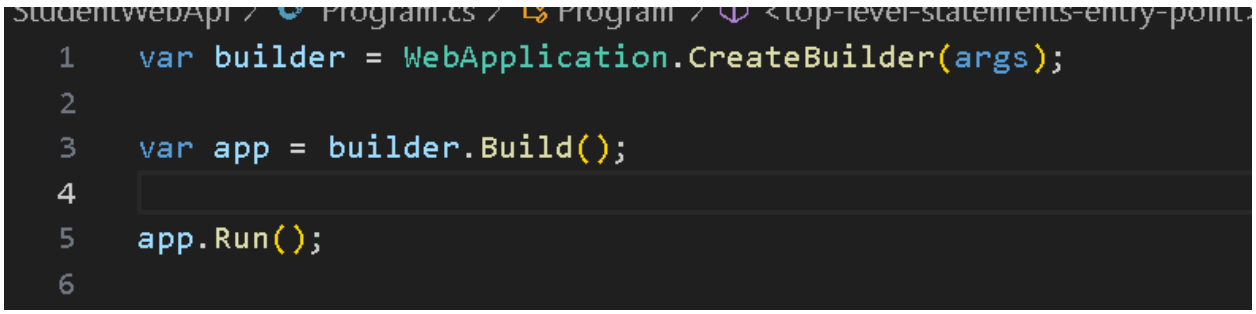
- **HTTP** = simple local testing → easy for tools like **Postman**.
- **HTTPS** = secure (encrypted) but requires extra certificates → not needed right now.
- Since we're just building & testing locally, **HTTP is enough** ✅.

🧹 Program.cs Cleanup

When a new project is generated, **Program.cs** comes with extra code like AddOpenApi, MapOpenApi, and the sample **WeatherForecast** API.

✅ What to retain:

- The **basic app startup** logic only



Keep this clean minimal code:

🛠️ Before Running

⚡ **Important:** Make sure your terminal path is inside the project folder.
For example:

C:\Users\jerem\Desktop\C#> cd StudentWebApi

PS C:\Users\jerem\Desktop\C#\StudentWebApi>

✅ Once your terminal shows that you're inside StudentWebApi, you're good to go.

🚀 Run the Project

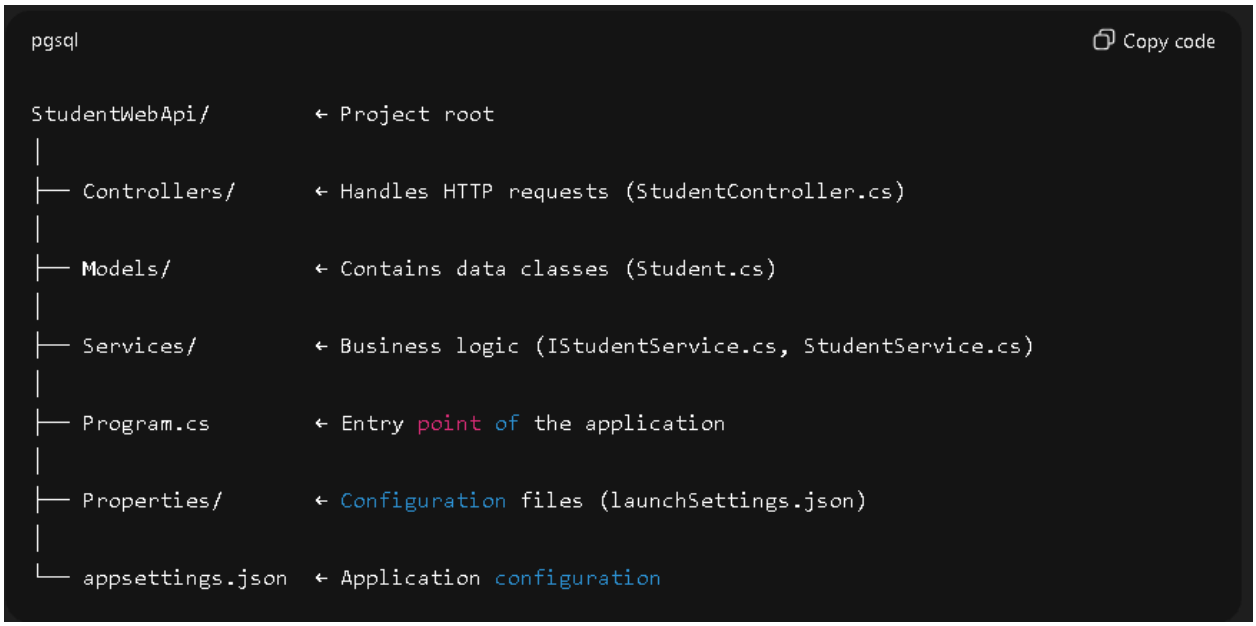
dotnet run

You should now see the output:

Now listening on: http://localhost:9090

📁 Project Structure & Naming Conventions

When building an ASP.NET Web API with layered architecture, we want our folders and files to be clear and consistent. Here's how we'll structure it:



🔑 Naming Conventions to Follow

- **Folders** → Controllers, Models, Services (PascalCase, plural).
- **Classes** → Student, StudentService, StudentController (PascalCase).
- **Interfaces** → Start with I → IStudentService.
- **Files** → Must match class/interface names (e.g., Student.cs for Student class).

🎯 Building Our First Model: Student

Why start with the **Model**?
👉 In software development, the **Model** represents the actual **data structure** of our application. For our case, this will be the **Student information**. Later, this object will be passed around between the **Service** and **Controller** layers.

📝 Step 1: Create the Models Folder

Inside your project folder structure:

📂 StudentWebApi
  ├ 📂 Models ← *(inside this folder)*
  ├ 📂 Services
  ├ 📂 Controllers
  └ Program.cs

Inside the **Models** folder:
➡ Right-click → New File → name it **Student.cs**

📝 Step 2: Add Attributes

Each **Student** needs to have:

- **StudentId (long)** → This acts as the Primary Key (PK), a unique identifier for each student.
- **Name (string)** → The name of the student.
- **Course (string)** → The course the student is enrolled in.

⚠️ Important:

- All attributes should be **private with controlled access** — this is **Encapsulation**, one of the four pillars of OOP.
- In **Java**, we usually write getName() and setName() methods.
- In **C#**, we use **Properties** with { get; set; } instead of writing separate methods.
    - Example: public string Name { get; set; }

📝 Step 3: Add Constructors

In C#, just like in Java, you can create **constructors** for initializing objects.

- **Default Constructor** → Creates an empty Student object.
- **Parameterized Constructor** → Allows you to create a Student object directly with StudentId, Name, and Course.

💡 Teacher's Note

👉 If you get confused about how get and set work in C#, take a quick lookup online. Compare **Java Getters/Setters** vs **C# Properties**. You'll notice that C# makes it shorter and cleaner.

⚙️ Service Layer – Applying Abstraction

❓ Why Service Layer?

In layered architecture:

- **Model** → Holds the data structure.
- **Service** → Contains the **business logic** (what your app can *do* with the data).
- **Controller** → Handles the API requests and responses.

The **Service Layer** is like the "brain" of the app — it decides how to manage the student data (add, retrieve, update, delete).

🔑 Step 1: Apply Abstraction

👉 In **C#**, we apply abstraction using **interfaces**.

- **Interface** = contract → only method signatures (no implementation).
- **Implementation Class** = the actual code for those methods.

💡 Real Life Example:

- Think of an **interface** as a **remote control** — it has buttons like Power, Volume, Channel (methods).
- The **TV** (implementation) decides *what happens* when you press those buttons.

📝 Step 2: Folder Setup

Inside your project folder structure:

```
📂 StudentWebApi
  ├ 📂 Models
  ├ 📂 Services
  |  ├ IStudentService.cs ← (Interface: defines methods)
  |  └ StudentService.cs ← (Implementation: contains logic)
  ├ 📂 Controllers
  └ Program.cs
```

📝 Step 3: Define the Interface (IStudentService)

Inside IStudentService.cs, you'll define abstract method signatures for the following operations:

- AddStudent(Student student)
- GetAllStudents()
- GetStudentById(long id)
- UpdateStudent(Student student)
- DeleteStudent(long id)

⚠️ No implementation here — just the method **signatures**.

📝 Step 4: Implement the Service (StudentService)

Inside StudentService.cs:

📝 How to Implement an Interface in C#

In **Java**, you'd write something like:

- **public class StudentService implements IStudentService { ... }**

In **C#**, the keyword **implements** does not exist. Instead, we just **use a colon :** after the class name:

👉 **C# way:**

- **public class StudentService : IStudentService**

💡 Think of the colon (:) as "this class **inherits or implements** something."

- If it's a class → it means inheritance (like : BaseClass).
- If it's an interface → it means implementation (like : IStudentService).

So in short:

- **Java** → implements keyword
- **C#** → : symbol

🛠️ Service Layer – Thought Process for Each Method

Inside StudentService, we'll simulate a database using a **Dictionary<long, Student>** (C# equivalent of Java's HashMap).

```csharp
Dictionary<long, Student> students = new Dictionary<long, Student>();
```

📋 Methods & Thought Process

- **getAllStudents()**
  Thought: Retrieve all values from the Dictionary and return as a List<Student>.

- **getStudentById(long id)**
  Thought: Check if the given ID exists in the Dictionary. If found, return the student. If not,

return null (since C# doesn't use Optional like Java).

- **addStudent(Student student)**
  Thought: Insert the Student into the Dictionary using pkStudentID as the key. Then return the same object to confirm it was added.

- **updateStudent(long id, Student student)**
  Thought: Check if a student with that ID exists. If yes → update the record and return the updated object. If no → return null or throw an exception.

- **deleteStudent(long id)**
  Thought: Check if the student exists in the Dictionary. If yes → remove it and return true. If not found → return false.

🛠️ Adjust Program.cs

Before creating the **Controller**, open **Program.cs** and make these changes:

1. **Add Controller Support**
   Insert:

   ```
   builder.Services.AddControllers();
   ```

2. **Map Controllers**
   Add:

   ```
   app.MapControllers();
   ```

📝 Explanation

- `builder.Services.AddControllers();` → Tells ASP.NET that we're using Controllers. Without this, your `[ApiController]` won't work.

- `app.MapControllers();` → Makes all the routes from your Controller (`/api/student/...`) available.

```csharp
var builder = WebApplication.CreateBuilder(args);

// Register controllers
builder.Services.AddControllers();

var app = builder.Build();

// Map all controller endpoints
app.MapControllers();

app.Run();
```

✨ Create the Controller File

- Inside your **Controllers** folder → create a new file called **StudentController.cs**.
- The file name should always end with *Controller* (naming convention).

✨ Add Attributes (in SpringBoot it's called Annotation)

At the top of your class, you'll add two key attributes:

```csharp
[ApiController]
[Route("api/[controller]")]
public class StudentController : ControllerBase
{
    // Your endpoints will go here
}
```

🔍 What do these mean?

- **`[ApiController]`**
  → Marks this class as a Web API controller. It gives you automatic features like request validation and proper response formatting.

- **`[Route("api/[controller]")]`**
  → Defines the base URL for this controller.
  If your class is named **StudentController**, then:

```bash
api/student
```

will be the root route for all endpoints inside this controller.

- **`ControllerBase`**

→ Instead of inheriting from `Controller` (used in MVC with views), Web APIs inherit from **ControllerBase** since we only return JSON data.

What is ControllerBase in ASP.NET Core?

- In ASP.NET, you have two main options for controllers:

  1. **`Controller`** → used in MVC (Model-View-Controller) apps, where you return *HTML views* + JSON.

  2. **`ControllerBase`** → used in Web API apps, where you return *only JSON or data responses*, not HTML.

`[HttpPost]`	Handles POST requests	`@PostMapping`
`[HttpPut]`	Handles PUT requests	`@PutMapping`
`[HttpDelete]`	Handles DELETE requests	`@DeleteMapping`