CREACIÓN DE UNA API Y UN CRUD MEDIANTE .NET

Dentro del proyecto (Carpeta), en el terminal:

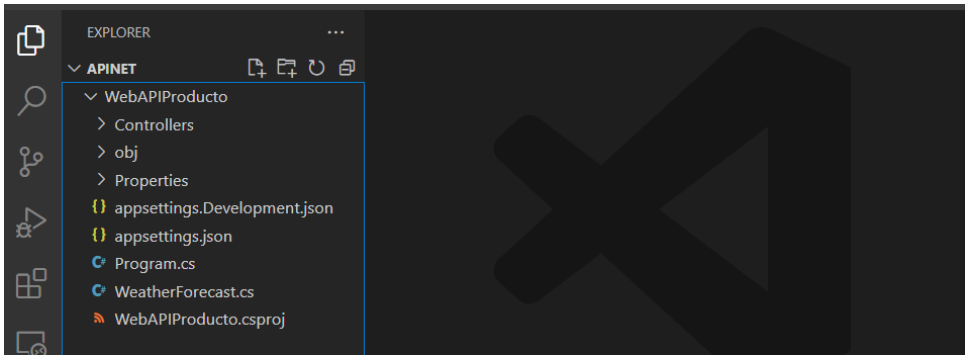
Para crear el directorio con el mismo nombre.

```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet>  dotnet new webapi -o WebAPIProducto
```

Para crear la carpeta dentro del directorio ya creado:

```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet>  dotnet new webapi -n WebAPIProducto
```

Pulsar intro y ya se tiene la estructura básica de una web API.



Se navega hacia la carpeta creada.

```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet> cd WebAPIProducto
```

Para mostrar los archivos del proyecto.

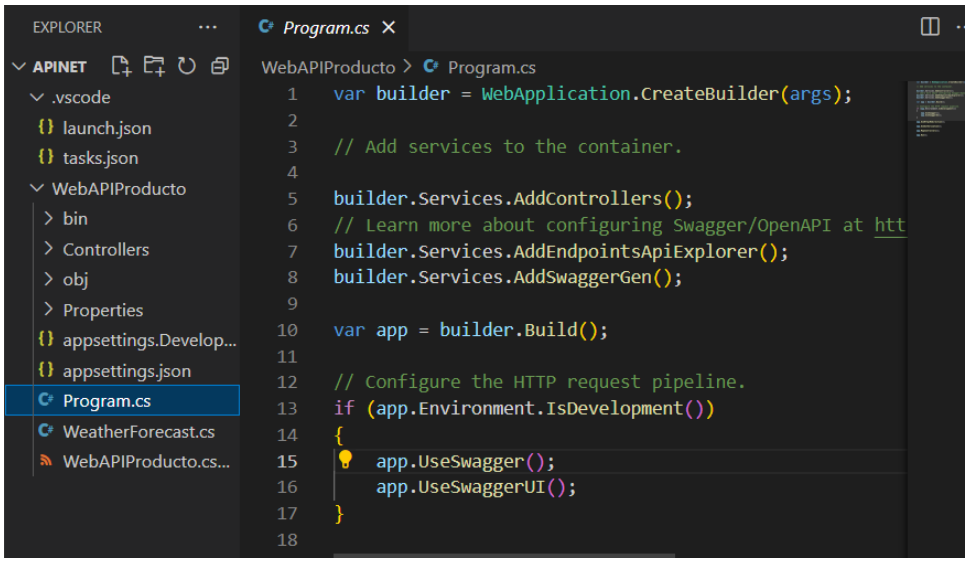
```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto> ls
```

```
PROBLEMS  OUTPUT  TERMINAL  ...        powershell

da---l        23/09/2023     17:30                Controllers
da---l        23/09/2023     17:30                obj
da---l        23/09/2023     17:30                Properties
-a---l        23/09/2023     17:30            127 appsettings.Develop
                                                   ment.json
-a---l        23/09/2023     17:30            151 appsettings.json
-a---l        23/09/2023     17:30            557 Program.cs
-a---l        23/09/2023     17:30            263 WeatherForecast.cs
-a---l        23/09/2023     17:30            408 WebAPIProducto.cspr
                                                   oj


PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto>
```

Estos son los mismo que se pueden ver en el visual en la parte izquierda, en el explorer.

Archivo del proyecto.

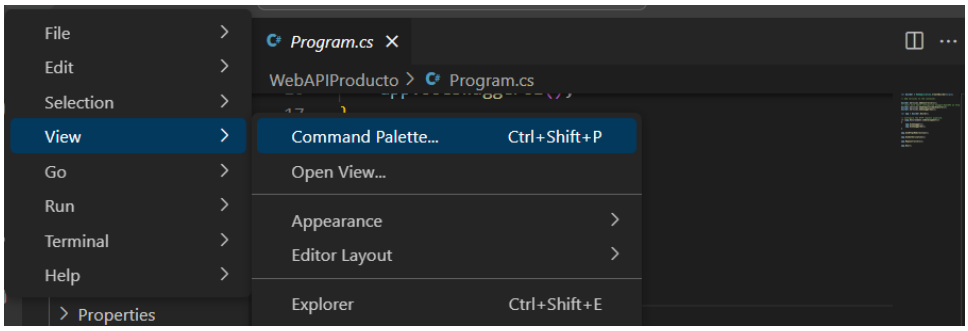
En el archivo Program.cs

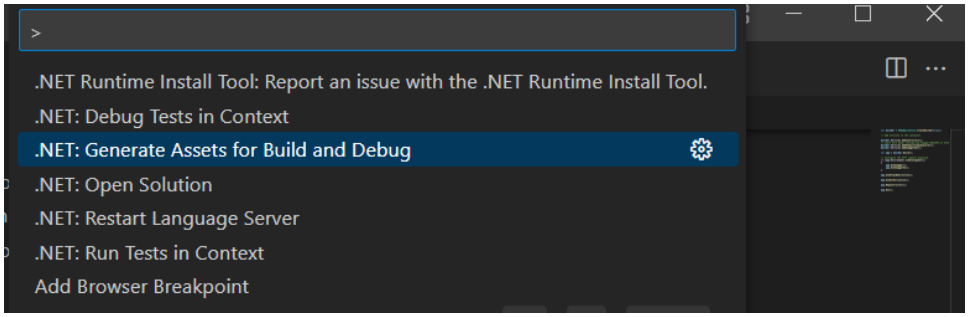


Dar a run a debbug, esto creara los siguientes archivos de configuración, los cuales, van a permitir ejecutar y compilar la aplicación.

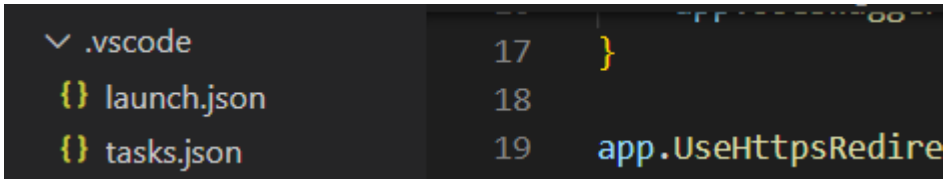
Además, que, si hay algún error, el editor lo marcara.

O bien en la paleta de comandos ver o View





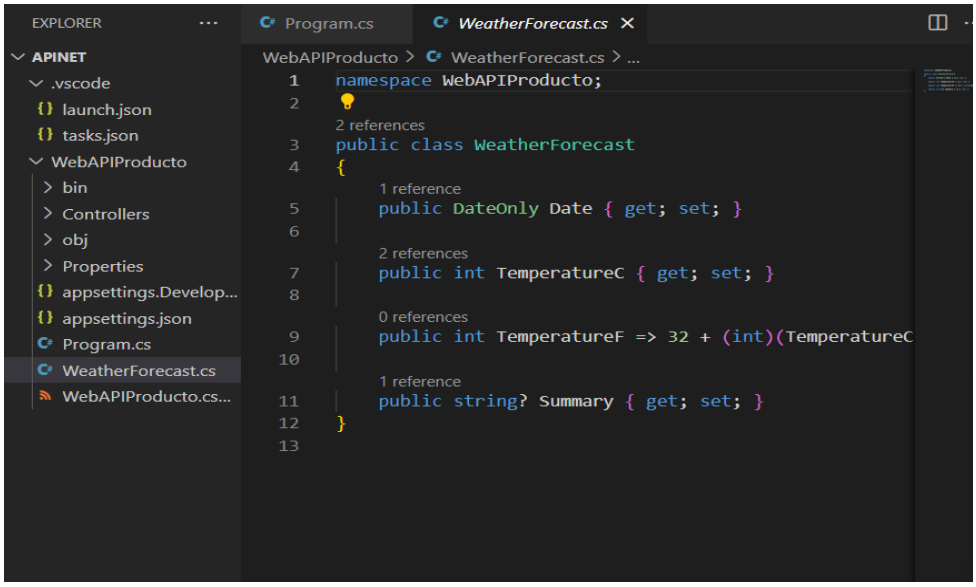
Y genera la carpeta de configuración .vscode.



En el archivo Program.cs estarían todas las configuraciones, como es el constructor de la aplicación, los servicios y la construcción de la app.

```
var builder = WebApplication.CreateBuilder(args);//constructor

// Add services to the container.

builder.Services.AddControllers();//servicios
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetco
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();//construcción de la App

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

app.UseHttpsRedirection();//miliwer

app.UseAuthorization();

app.MapControllers();//mapear controladores

app.Run();// correr la aplicación.
```

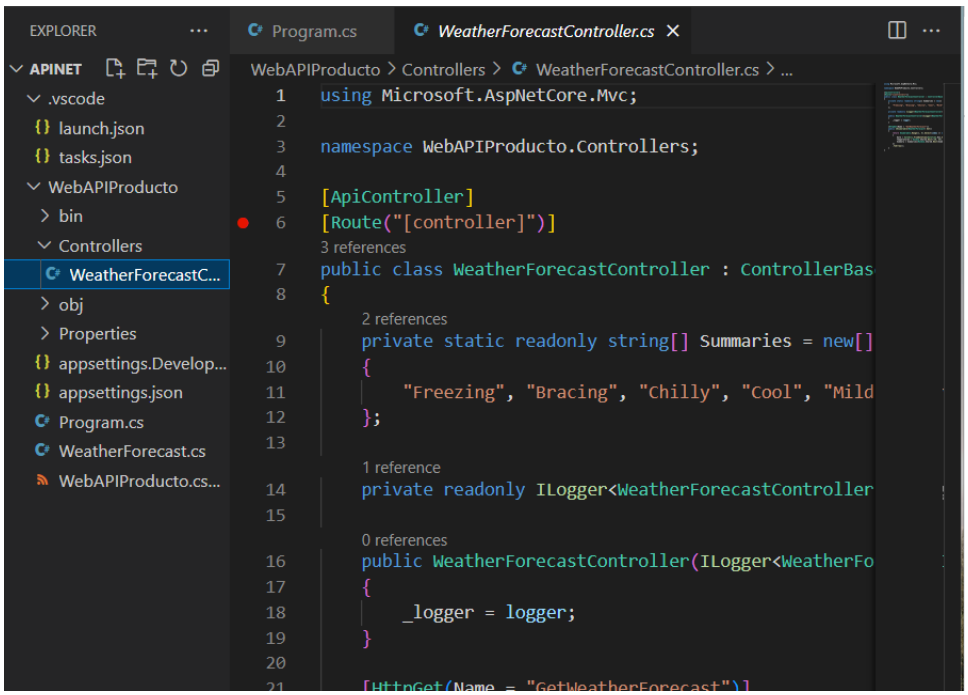
Clase de prueba, del tiempo y la temperatura.



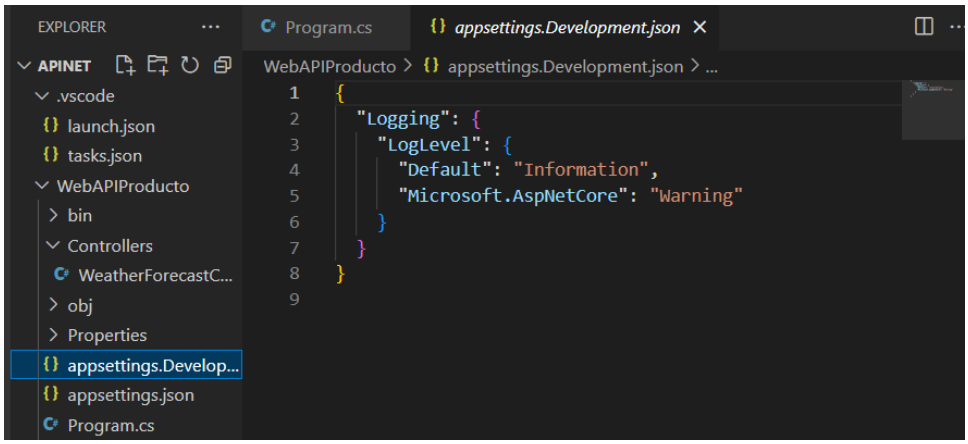
En la carpeta controladores es donde se crea el controlador para los productos, ya está el controlador de tipo **[ApiController]**, es lo necesario para crear los métodos y que sean tratados como servicio rest.

[Route("[controller]")] va a permitir como indicar la ruta de ese método.

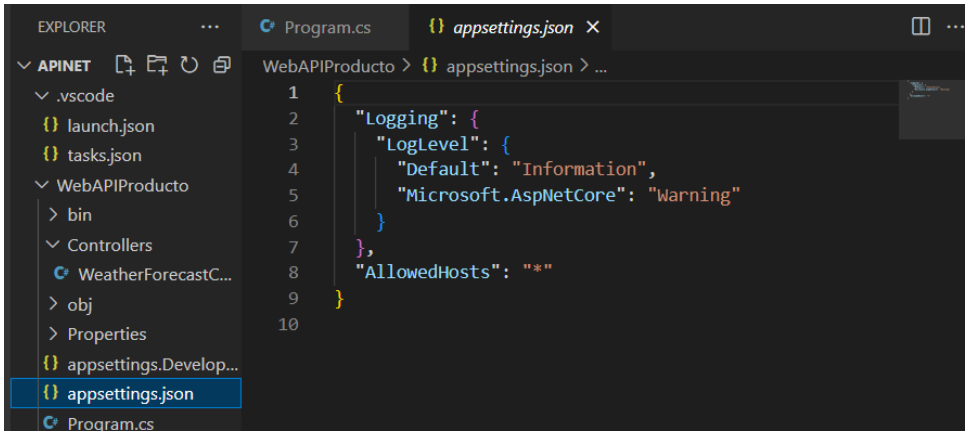
Public class WeatherForecastController:ControllerBase es el nombre del controlador, normalmente es el nombre del controlador y el sufijo después de los dos puntitos.



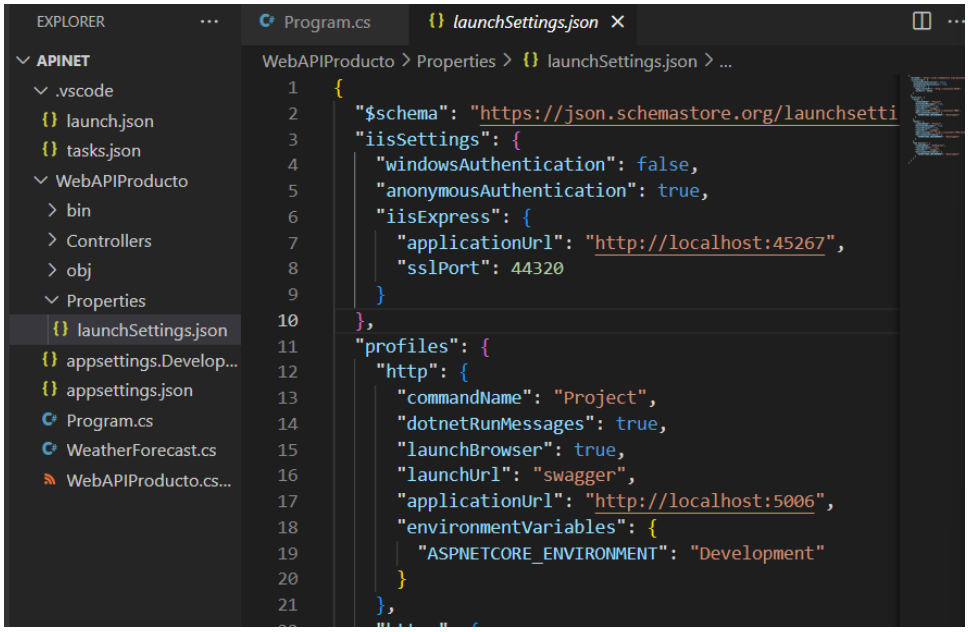
Configuraciones para el caso de desarrollo.



Para la publicación de nuestro web api.



En la carpeta propiedades se puede ver el puerto en el que se está ejecutando



Ya comentada la estructura básica de un proyecto.

Para la ejecución del proyecto.

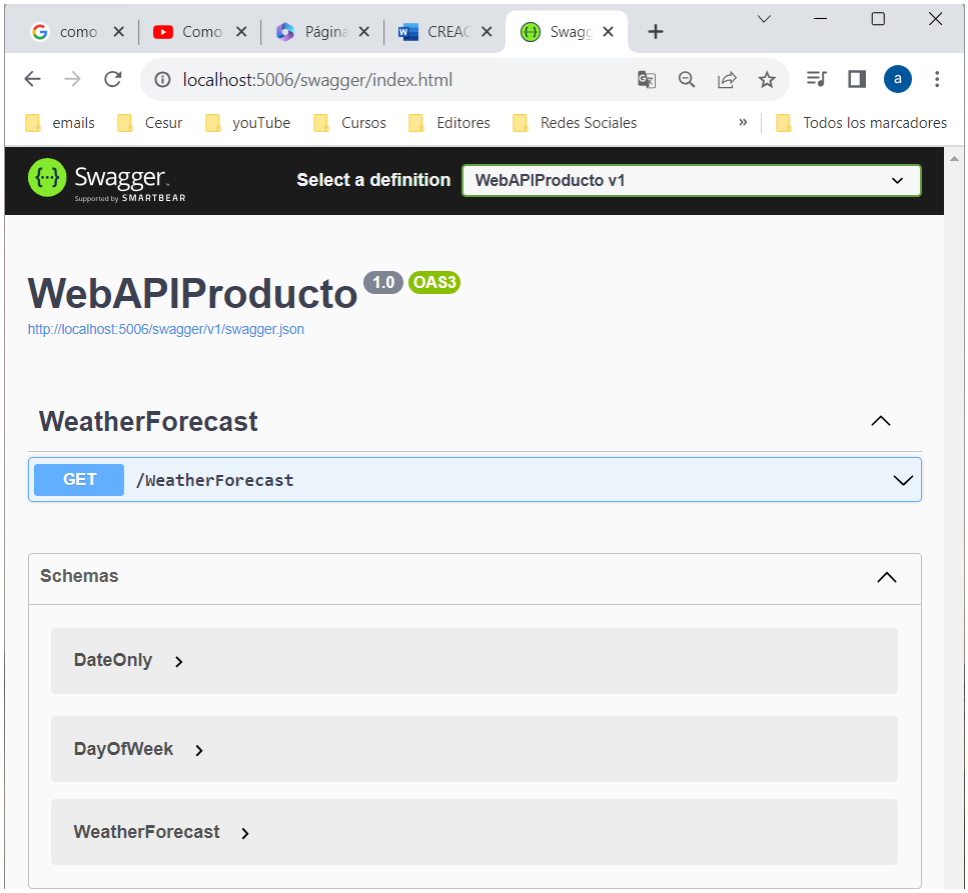
```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto> dotnet run
```

En la URL que ofrece por consola, se hace clic en ella.

```
    Now listening on: http://localhost:5006
nfo: Microsoft.Hosting.Lifetime[0]
```

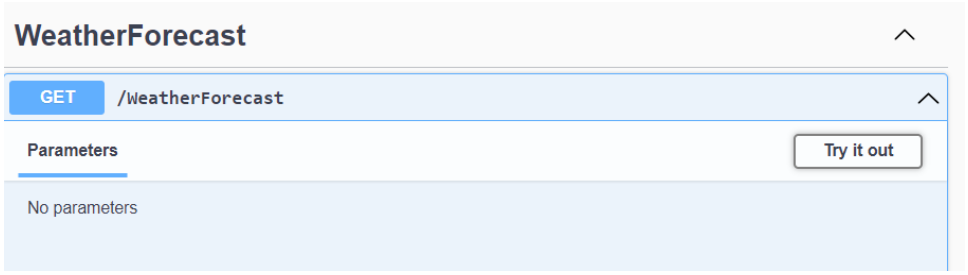
Acto seguido se abre el navegador y despues de localhost:5006, se añade la palabra swagger llevando a la página.

http://localhost:5006/swagger

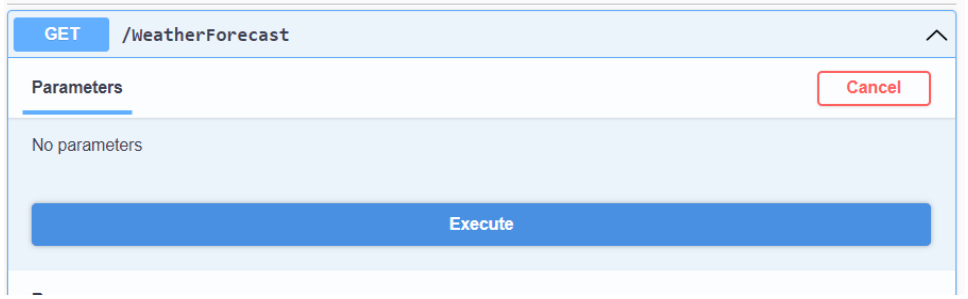


Esta página lo que hace es mapear y crear la documentación de los métodos que tenemos en nuestra web.

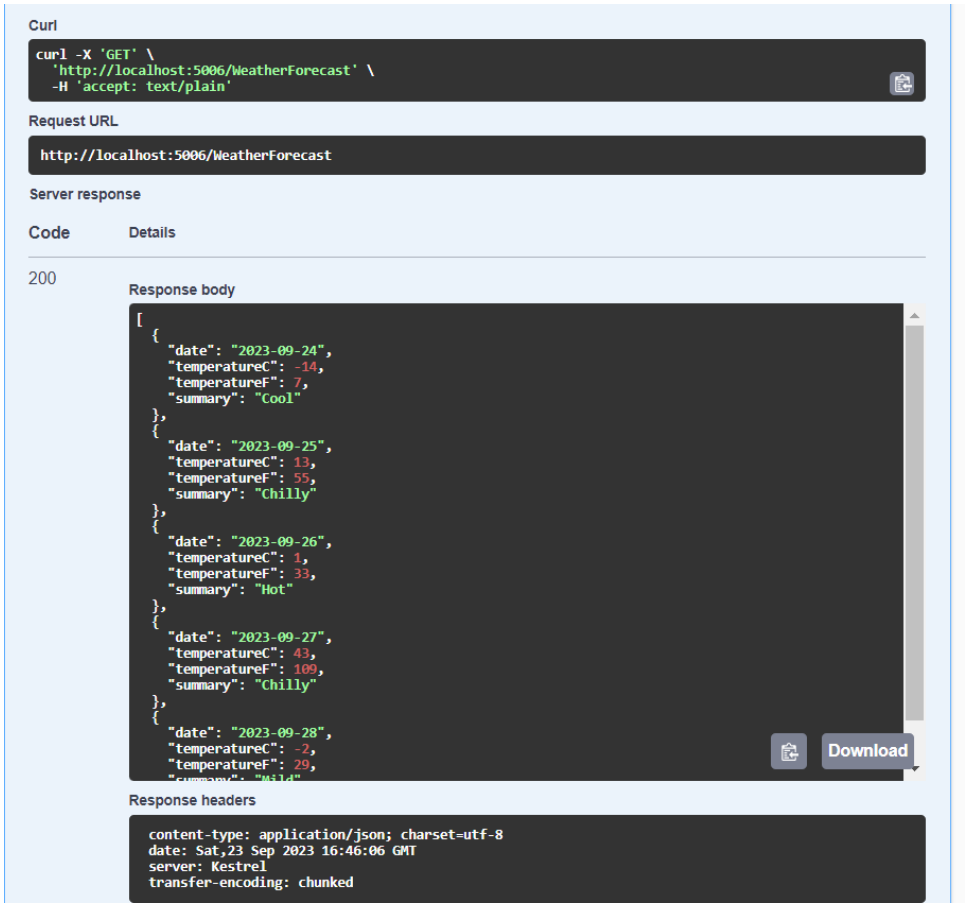
Ahora mismo solo hay un método.



Si seleccionamos Try it out y ejecutamos.



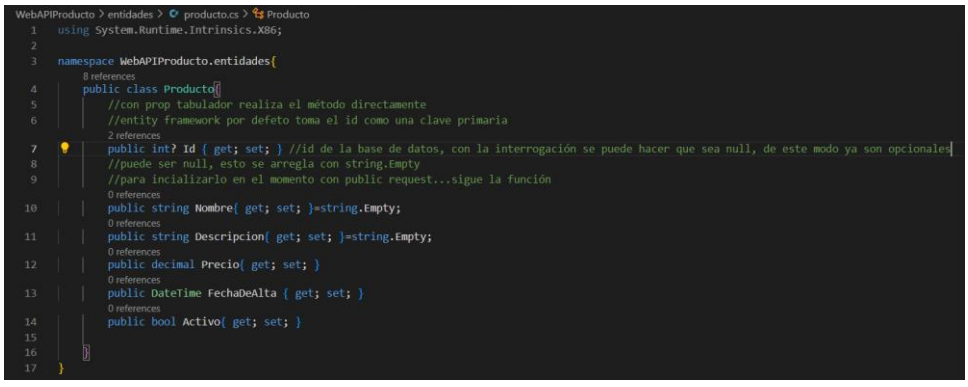
Devuelve la respuesta en formato JSON, de los elementos weatherForecast.



Se detiene la aplicación con ctr+c en el terminal.

En el programa de visual estudio code dentro de la carpeta WebAPIProducto, se crea una carpeta para las entidades.

Se realiza la creación de la primera clase, en este caso la clase producto.

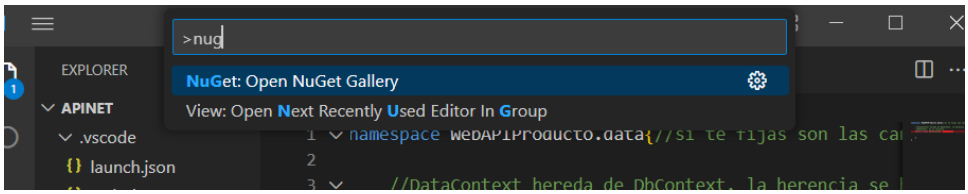


Una vez completado el modelo, el siguiente paso es agregar el div inconter para conectarse a la base de datos.

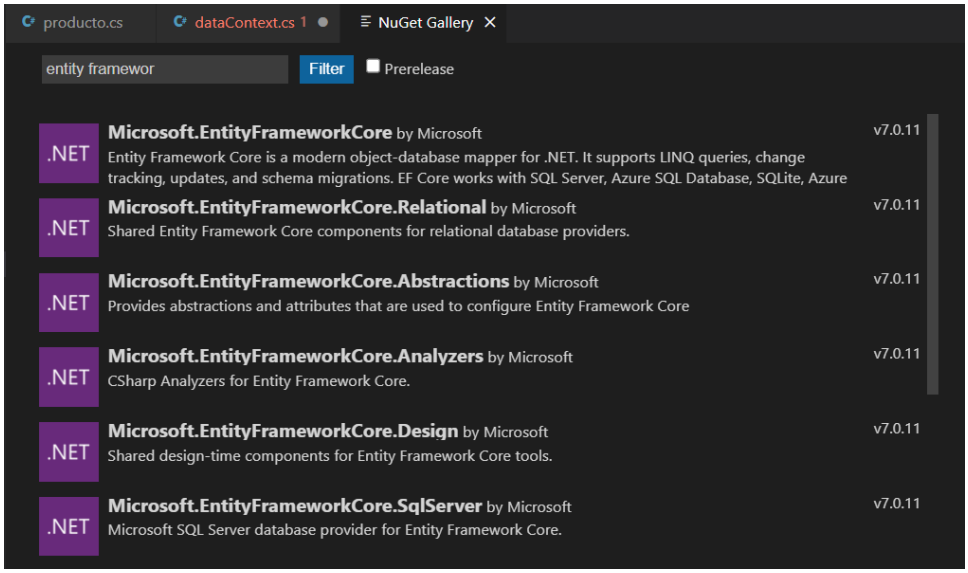
En el programa de visual estudio code dentro de la carpeta WebAPIProducto, se crea una carpeta para la conexión, en este caso la carpeta data.

Se crea el archivo ‘dataContexts.css’.

DbContext no lo reconoce, en vista paleta de comandos:

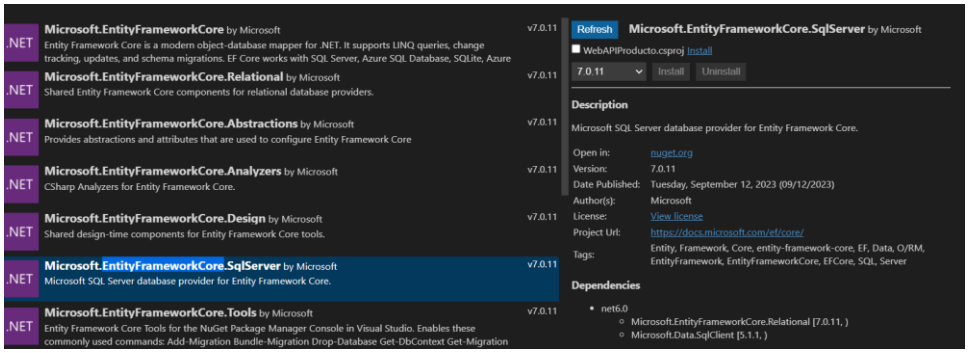


Intro y se busca el framework entity.

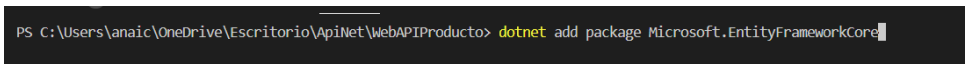


Aquí aparecen todos y en este caso se selecciona SqlServer.

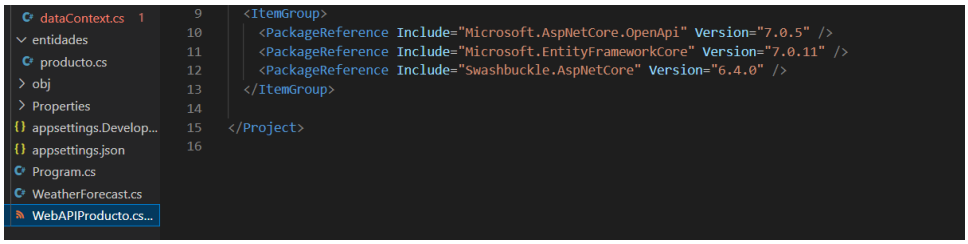
La instalación se puede hacer desde visual estudio code.



O bien desde consola. dotnet add package Microsoft.EntityFrameworkCode

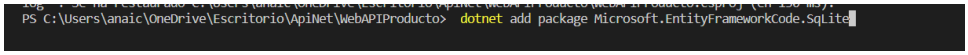


Se puede observar que en el archivo WebAPIProducto.cs se añadido los paquetes.

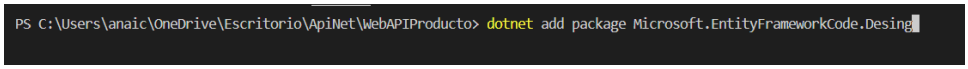


También se instala los paquetes para trabajar con SqLite.

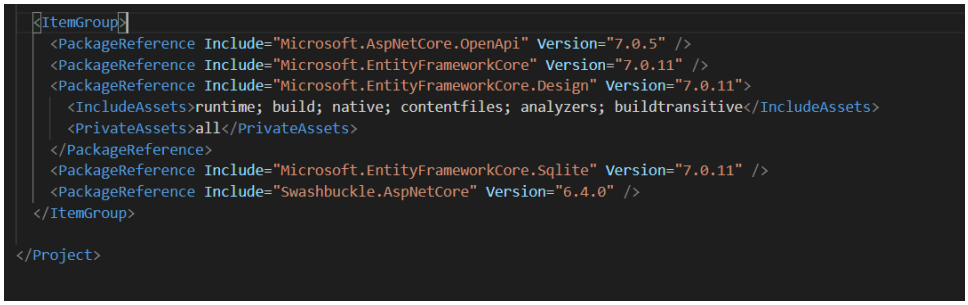
dotnet add package Microsoft.EntityFrameworkCode.SqLite



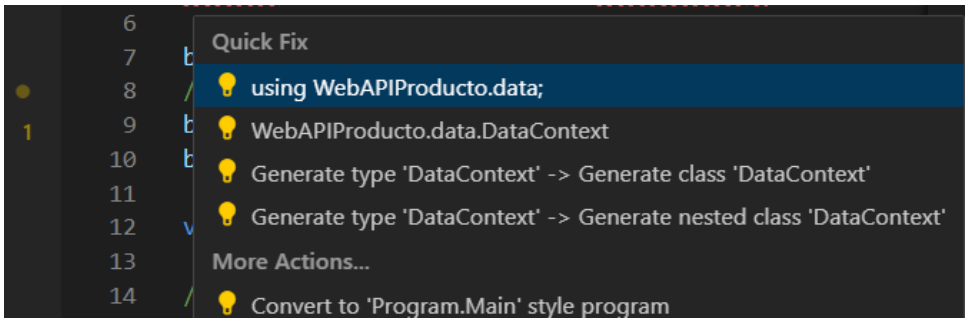
Cuando empecemos a trabajar con migraciones también pedirá el paquete Design.
dotnet add package Microsoft.EntityFrameworkCode.Design



Ya añadidos los paquetes.



Volvemos al archivo ‘dataContext.css’, en la bombilla sale un mesange usin, es este el que se selecciona. Y ya sí que reconocería DbContext.



Ya se puede terminar de realizar el modelo.

```
namespace WebAPIProducto.data{//si te fijas son las carpetas donde se encuentra

    //DataContext hereda de DbContext, la herencia se hace con los dos puntitos
    //DbContext, no lo reconoce
    2 references
    public class DataContext: DbContext{
        //Se agrega el constructor poner ctor y tabulador (lo crea directamente)
        //DbContextOptions contexto en el que nos encontramos (DbContext)
        //<DataContext>El tipo que recibe por paárametro (Es la clase)
        //option variable
        //:base(options)pasa la variable a la clase base
        //La clase base es la que va ha realizar la implementación (realiza el trabajo)

        0 references
        public DataContext(DbContextOptions<DataContext>options):base(options)
        {
        }
        //Creara una BD del producto que estoy realizando, se llamará producto de tipo <producto>
        //el nombre que se le ponga ese será el nombre de la tabla, en este caso se llama Producto

        0 references
        public DbSet<Producto> Producto {get; set;}
    }
}
```

Una vez terminado el modelo hay que registrarlo en la inyección de independencia, que se encuentra en el archivo 'Program.cs'

Se añade el código

```
using Microsoft.EntityFrameworkCore;
using WebAPIProducto.data;

var builder = WebApplication.CreateBuilder(args);//constructor

// Add services to the container.
//para agregar las independencias para la conexión
//en UseSqlite()le entra la cadena de conexión a la BD
builder.Services.AddDbContext<DataContext>(options=>options.UseSqlite());

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

La cadena de conexión se agrega en el archivo 'appsettings.json' ya que es el archivo de configuración.

```
WebAPIProducto > {} appsettings.json > {} Logging > {} LogLevel
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection":"Data Source=DataApi.db"
  }

}
```

Y ya se tiene la cadena de conexión, esta se guarda en una variable.

```
using Microsoft.EntityFrameworkCore;
using WebAPIProducto.data;

var builder = WebApplication.CreateBuilder(args);//constructor

// Add services to the container.
//para agregar las independencias para la conexión
//en UseSqlite()le entra la cadena de conexión a la BD
var cadenaConexion = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<DataContext>(options=>options.UseSqlite(cadenaConexion));
```

Ya se tiene la configuración para usar la base de datos. El siguiente paso es crear las migraciones (crear las tablas si no exite).

Hay que instalar la herramienta. dotnet tool install --global dotnet-ef

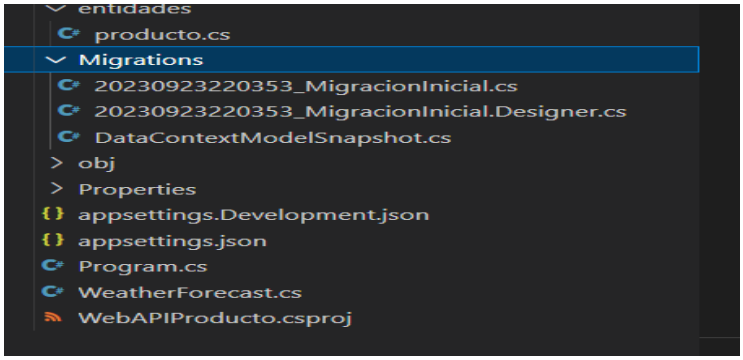
```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto> dotnet tool install --global dotnet-ef
```

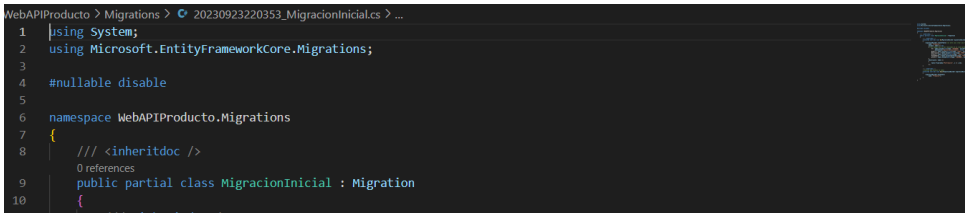
```
Puede invocar la herramienta con el comando siguiente: dotnet-ef
La herramienta "dotnet-ef" (versión '7.0.11') se instaló correctamente.
```

Se realiza la migración. dotnet ef migrations add Nommigracion(el que se quiera)

```
dotnet ef migrations add MigracionInicial
```

Crea los archivos





```
/// <inheritdoc />
0 references
protected override void Up(MigrationBuilder migrationBuilder)//método que va a escribir o hacer cambios en la BD
{
    migrationBuilder.CreateTable(//en este caso crear una tabla
        name: "Producto",
        columns: table => new
        {//los campos que se han realizado en la clase producto
            Id = table.Column<int>(type: "INTEGER", nullable: false)
                .Annotation("Sqlite:Autoincrement", true),
            Nombre = table.Column<string>(type: "TEXT", nullable: false),
            Descripcion = table.Column<string>(type: "TEXT", nullable: false),
            Precio = table.Column<decimal>(type: "TEXT", nullable: false),
            FechaDeAlta = table.Column<DateTime>(type: "TEXT", nullable: false),
            Activo = table.Column<bool>(type: "INTEGER", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Producto", x => x.Id);
        });
}
```

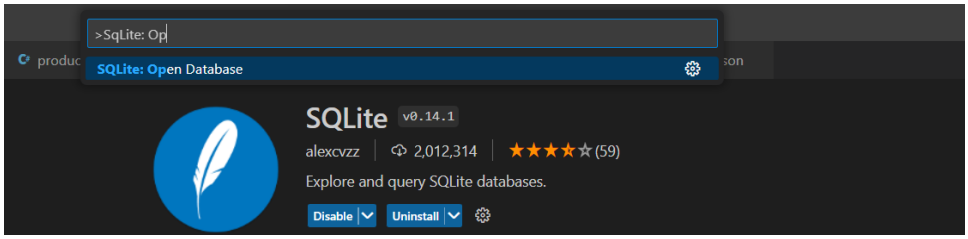
```
/// <inheritdoc />
//función que elimina la tabla
0 references
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Producto");
}
}
}
```

Una vez ya se tiene la tabla, hay que crear la base de datos, esto se hace con el comando: dotnet ef database update

```
dotnet ef database update
```

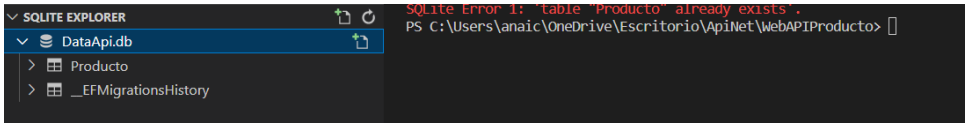
Ya aparece la base de datos creada en la parte izquierda del programa.

Ahora con la extensión SqLite.



Se abre la Bd, al dar para abrir crea los siguientes archivos.

Se puede ver la base de datos en SQULITE EXPLORER



Ahora se genera el **controlador** (CRUD).

En el archivo Controllers, un archivo, en este caso 'ProductoController.cs'

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using WebAPIProducto.data;
using WebAPIProducto.entidades;

namespace WebAPIProducto.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    3 references
    public class ProductoController : ControllerBase
    {
        //para registrar los eventos
        1 reference
        private readonly ILogger<ProductoController> _logger;
        //Contexto
        3 references
        private readonly DataContext _context;
        //constructor
        0 references
        public ProductoController(ILogger<ProductoController> logger,DataContext context ){
            _logger=logger;
            _context=context;

        }
```

```
//creación de método getAll (Los devuelve todos los registros)
[HttpGet(Name = "GetProductos")]
//traer lo productos de tipo asincrona (async)
0 references
public async Task<ActionResult<IEnumerable<Producto>>> GetProductos() {
    var productos = await _context.Producto.ToListAsync();
    return Ok(productos);
}
//método get solo va a devolver un registro
[HttpGet("{id}",Name = "GetProducto")]

0 references
public async Task<ActionResult<Producto>> GetProducto(int id) {
    var productos = await _context.Producto.FindAsync(id);//Este método encuentra el producto que coincide con el id
    if(productos==null){
        return NotFound();
    }
    return productos;

}
```

```
//método para añadir un registro en la tabla
[HttpPost]
0 references
public async Task<ActionResult<Producto>> Post(Producto producto) {
    _context.Add(producto);
    await _context.SaveChangesAsync();
return new CreatedAtRouteResult ("GetProducto", new{Id=producto.Id},producto);
}
//método para actualizar (modificar) en la tabla
[HttpPut]
0 references
public async Task<ActionResult<Producto>> Put(int id,Producto producto) {
 if(id != producto.Id){
   return BadRequest();
 }
 _context.Entry(producto).State = EntityState.Modified;
 await _context.SaveChangesAsync();
return Ok();

}
```

```
    //método para eliminar registro de la tabla
    [HttpDelete("{id}")]
    0 references
    public async Task<ActionResult<Producto>>Delete(int id){
        var producto = await _context.Producto.FindAsync(id);
        if(producto == null){
            return NotFound();
        }
        _context.Producto.Remove(producto);
        await _context.SaveChangesAsync();
        return producto;
    }
}
}
```

Ya con los métodos para el CRUD realizados, el siguiente paso es compilar y ejecutar. Con el comando dotnet run

```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto> dotnet run
```

Ahora la tabla está vacía por lo que con el método PUT se crean registros.