

CREACIÓN DE UNA API Y UN CRUD MEDIANTE .NET

Dentro del proyecto (Carpeta), en el terminal:

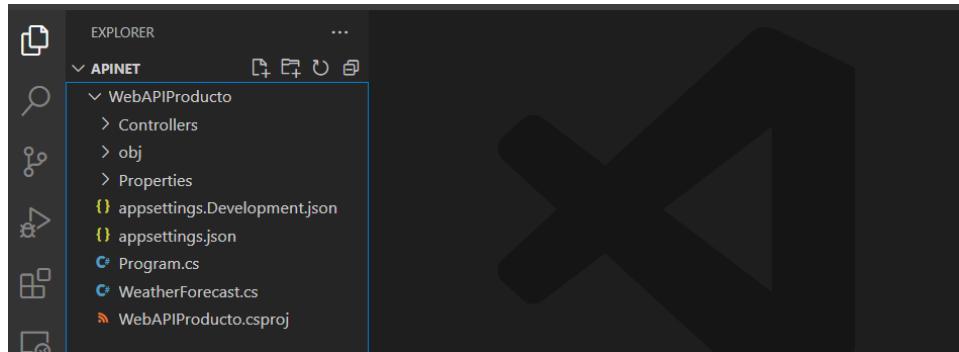
Para crear el directorio con el mismo nombre.

```
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							
da---l	23/09/2023	17:30							
da---l	23/09/2023	17:30							
-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.



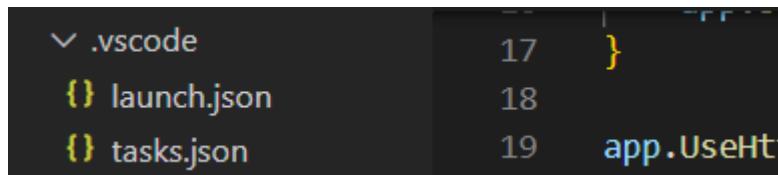
En el archivo Program.cs

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at http://tinyurl.com/8sBx8m
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

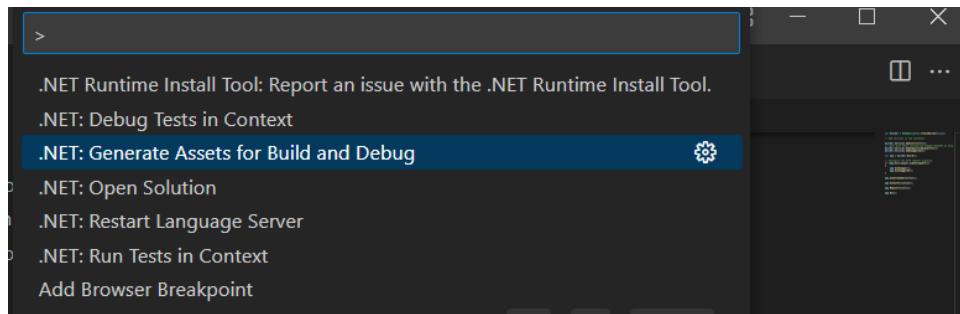
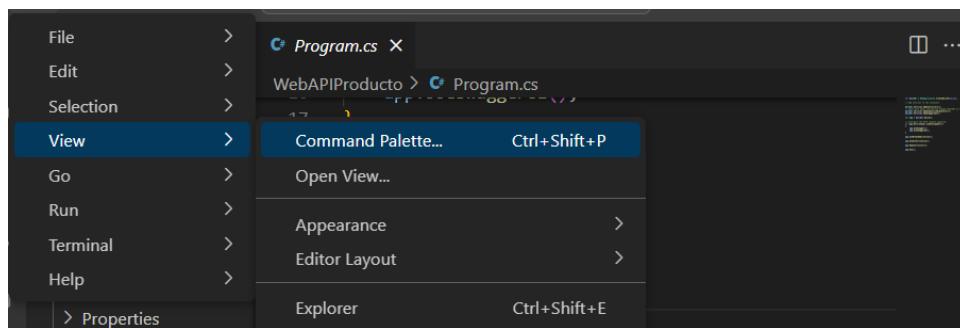
var app = builder.Build();
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Dar a run a debug, 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 marcará.



O bien en la paleta de comandos ver o View



Y genera la carpeta de configuración .vscode.

```

    .vscode
    { } launch.json
    { } tasks.json

```

```

17     }
18
19     app.UseHttpsRedire

```

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

```

1 var builder = WebApplication.CreateBuilder(args); //constructor
2
3 // Add services to the container.
4
5 builder.Services.AddControllers(); //servicios
6 // Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore-api-configuration
7 builder.Services.AddEndpointsApiExplorer();
8 builder.Services.AddSwaggerGen();
9
10 var app = builder.Build(); //construcción de la App
11
12 // Configure the HTTP request pipeline.
13 if (app.Environment.IsDevelopment())
14 {
15     app.UseSwagger();
16     app.UseSwaggerUI();
17 }
18
19 app.UseHttpsRedirection(); //miliwer
20
21 app.UseAuthorization();
22
23 app.MapControllers(); //mapear controladores
24
25 app.Run(); // correr la aplicación.
26

```

Clase de prueba, del tiempo y la temperatura.

```

EXPLORER      ...
APINET
  .vscode
    { } launch.json
    { } tasks.json
  WebAPIProducto
    bin
    Controllers
    obj
    Properties
    { } appsettings.Develop...
    { } appsettings.json
    Program.cs
    WeatherForecast.cs
    WebAPIProducto.cs...

```

```

Program.cs      WeatherForecast.cs
WebAPIProducto > WeatherForecast.cs > ...
1 namespace WebAPIProducto;
2
3 public class WeatherForecast
4 {
5     public DateOnly Date { get; set; }
6
7     public int TemperatureC { get; set; }
8
9     public int TemperatureF => 32 + (int)(TemperatureC * 1.8);
10
11    public string? Summary { get; set; }
12
13

```

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.

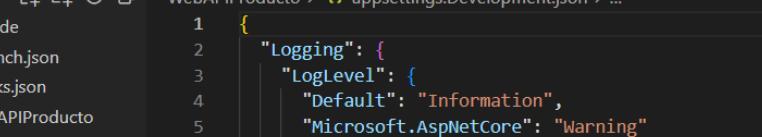
Public class WeatherForecastController:ControllerBase es el nombre del controlador, normalmente es el nombre del controlador y el sufijo después de los dos puntos.

The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER** pane on the left, showing the project structure:
 - APINET
 - .vscode
 - launch.json
 - tasks.json
 - WebAPIProducto
 - bin
 - Controllers
 - WeatherForecastC... (selected)
 - obj
 - Properties
 - appsettings.Develop...
 - appsettings.json
 - Program.cs
 - WeatherForecast.cs
 - WebAPIProducto.cs...
- Editor** pane on the right, showing the code for `WeatherForecastController.cs`:

```
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace WebAPIProducto.Controllers;
4
5  [ApiController]
6  [Route("[controller]")]
7  public class WeatherForecastController : ControllerBase
8  {
9      private static readonly string[] Summaries = new[]
10     {
11         "Freezing", "Bracing", "Chilly", "Cool", "Mild",
12     };
13
14     private readonly ILogger<WeatherForecastController>
15
16     public WeatherForecastController(ILogger<WeatherFo
17     {
18         _logger = logger;
19     }
20
21     [HttpGet(Name = "GetWeatherForecast")]
22 }
```

Configuraciones para el caso de desarrollo.



```
1  {
2      "Logging": {
3          "LogLevel": {
4              "Default": "Information",
5              "Microsoft.AspNetCore": "Warning"
6          }
7      }
8  }
9
```

Para la publicación de nuestro web api.

```
1  {
2      "Logging": {
3          "LogLevel": {
4              "Default": "Information",
5              "Microsoft.AspNetCore": "Warning"
6          }
7      },
8      "AllowedHosts": "*"
9  }
```

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

```
1  {
2      "$schema": "https://json.schemastore.org/launchsettings.json",
3      "iisSettings": {
4          "windowsAuthentication": false,
5          "anonymousAuthentication": true,
6          "iisExpress": {
7              "applicationUrl": "http://localhost:45267",
8              "sslPort": 44320
9          }
10     },
11     "profiles": {
12         "http": {
13             "commandName": "Project",
14             "dotnetRunMessages": true,
15             "launchBrowser": true,
16             "launchUrl": "swagger",
17             "applicationUrl": "http://localhost:5006",
18             "environmentVariables": {
19                 "ASPNETCORE_ENVIRONMENT": "Development"
20             }
21         }
22     }
23 }
```

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
info: 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>

The screenshot shows a browser window with the address bar pointing to `localhost:5006/swagger/index.html`. The main content is the Swagger UI for a 'WebAPIProducto v1' API. At the top, there's a navigation bar with tabs for 'Como', 'Como', 'Página', 'CREAC', and 'Swagger'. The 'Swagger' tab is active. Below the navigation is a search bar and a sidebar with links like 'emails', 'Cesur', 'youtube', 'Cursos', 'Editores', 'Redes Sociales', and 'Todos los marcadores'. The main area has a dark header with the 'Swagger' logo and 'Supported by SMARTBEAR'. A dropdown menu 'Select a definition' is set to 'WebAPIProducto v1'. The main title is 'WebAPIProducto 1.0 OAS3'. Below it is the URL `http://localhost:5006/swagger/v1/swagger.json`. A large section titled 'WeatherForecast' contains a 'GET' button and a URL field with `/WeatherForecast`. To the right of this is a 'Try it out' button. Below this is a 'Schemas' section with three items: 'DateOnly >', 'DayOfWeek >', and 'WeatherForecast >'. Each item has a small arrow icon to its right.

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.

This screenshot shows the 'WeatherForecast' endpoint in more detail. The title is 'WeatherForecast' and the method is 'GET' with the URL `/WeatherForecast`. To the right is a 'Try it out' button. Below this is a 'Parameters' section which says 'No parameters'. There is also a 'Cancel' button.

Si seleccionamos Try it out y ejecutamos.

This screenshot shows the 'WeatherForecast' endpoint being executed. The title is 'WeatherForecast' and the method is 'GET' with the URL `/WeatherForecast`. The 'Parameters' section says 'No parameters'. Below this is a large blue 'Execute' button. To the right is a 'Cancel' button.

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

The screenshot shows a network request for 'http://localhost:5006/WeatherForecast'. The response status is 200 OK. The response body is a JSON array containing five weather forecast entries. Each entry includes a date, temperature in Celsius and Fahrenheit, and a summary. The response headers indicate the content type is application/json and the date is Sat, 23 Sep 2023 16:46:06 GMT.

```
curl -X 'GET' \
'http://localhost:5006/WeatherForecast' \
-H 'accept: text/plain'

Request URL
http://localhost:5006/WeatherForecast

Server response

Code Details
200 Response body
[{"date": "2023-09-24", "temperatureC": -14, "temperatureF": 7, "summary": "Cool"}, {"date": "2023-09-25", "temperatureC": 13, "temperatureF": 55, "summary": "Chilly"}, {"date": "2023-09-26", "temperatureC": 1, "temperatureF": 33, "summary": "Hot"}, {"date": "2023-09-27", "temperatureC": 43, "temperatureF": 109, "summary": "Chilly"}, {"date": "2023-09-28", "temperatureC": -2, "temperatureF": 29, "summary": "Mild"}]

Response headers
content-type: application/json; charset=utf-8
date: Sat, 23 Sep 2023 16:46:06 GMT
server: Kestrel
transfer-encoding: chunked
```

Se detiene la aplicación con **ctrl+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.

The screenshot shows the code editor for 'producto.cs' in the 'entities' folder. The class 'Producto' is defined with properties: Id (nullable int), Nombre (string), Descripcion (string), Precio (decimal), FechaDeAlta (DateTime), and Activo (bool). The code uses nullable types and string.Empty for empty strings.

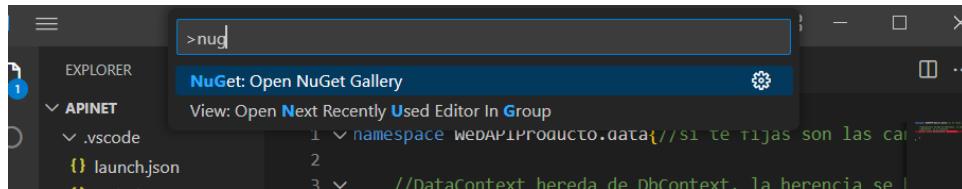
```
WebAPIProducto > entidades > producto.cs > Producto
1  using System.Runtime.Intrinsics.X86;
2
3  namespace WebAPIProducto.entidades{
4      public class Producto{
5          //con prop tabulador realiza el método directamente
6          //entity framework por defecto toma el id como una clave primaria
7          public int? Id { get; set; } //id de la base de datos, con la interrogación se puede hacer que sea null, de este modo ya son opcionales
8          //puede ser null, esto se arregla con string.Empty
9          //para inicializarlo en el momento con public request...sigue la función
10         public string Nombre{ get; set; }=string.Empty;
11         public string Descripcion{ get; set; }=string.Empty;
12         public decimal Precio{ get; set; }
13         public DateTime FechaDeAlta { get; set; }
14         public bool Activo{ get; set; }
15     }
16 }
```

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

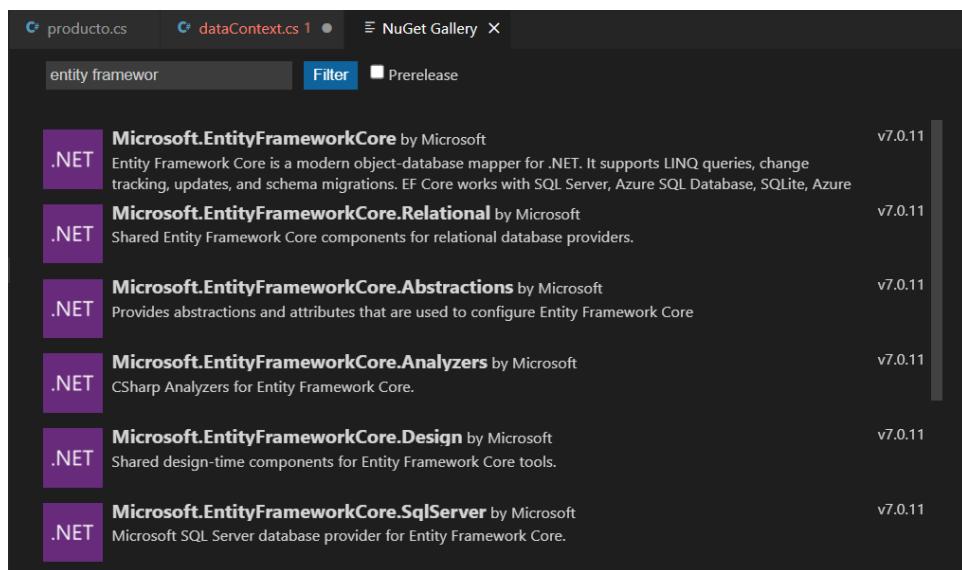
En el programa de visual estudio code dentro de la carpeta WebAPIProducto, se crea una carpeta para la conexión, en este caso la carpeta data.

Se crea el archivo 'dataContexts.css'.

DbContext no lo reconoce, en vista paleta de comandos:

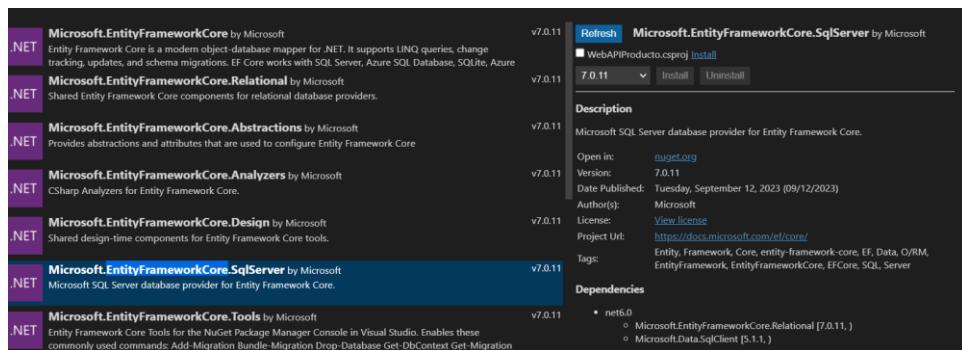


Intro y se busca el framework entity.



Aquí aparecen todos y en este caso se selecciona SqlServer.

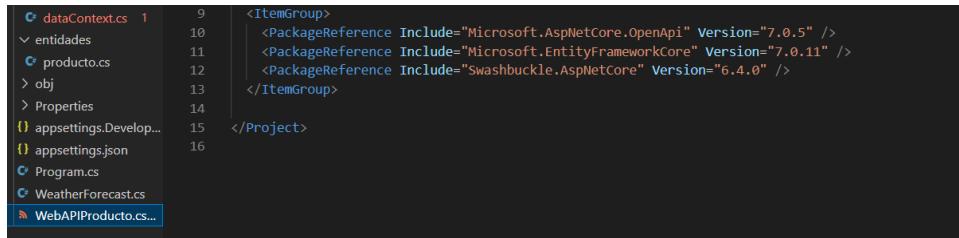
La instalación se puede hacer desde visual estudio code.



O bien desde consola. dotnet add package Microsoft.EntityFrameworkCore

```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto> dotnet add package Microsoft.EntityFrameworkCore
```

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



```
 1 9      <ItemGroup>
 2 10     <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="7.0.5" />
 3 11     <PackageReference Include="Microsoft.EntityFrameworkCore" Version="7.0.11" />
 4 12     <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
 5 13   </ItemGroup>
 6 14 </Project>
 7 15 <appsettings.Develop...
 8 16 <appsettings.json>
 9 17 <Program.cs>
10 18 <WeatherForecast.cs>
11 19 <WebAPIProducto.cs...>
```

También se instala los paquetes para trabajar con SQLite.

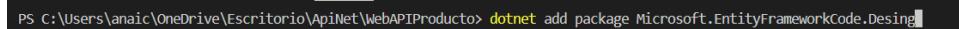
```
dotnet add package Microsoft.EntityFrameworkCore.SQLite
```



```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto> dotnet add package Microsoft.EntityFrameworkCore.SQLite
```

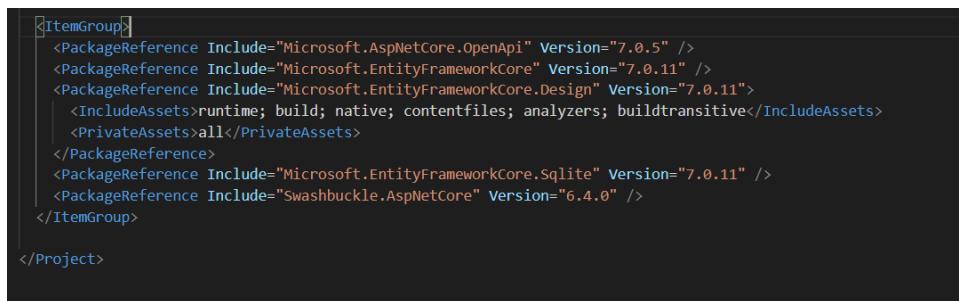
Cuando empecemos a trabajar con migraciones también pedirá el paquete Design.

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```



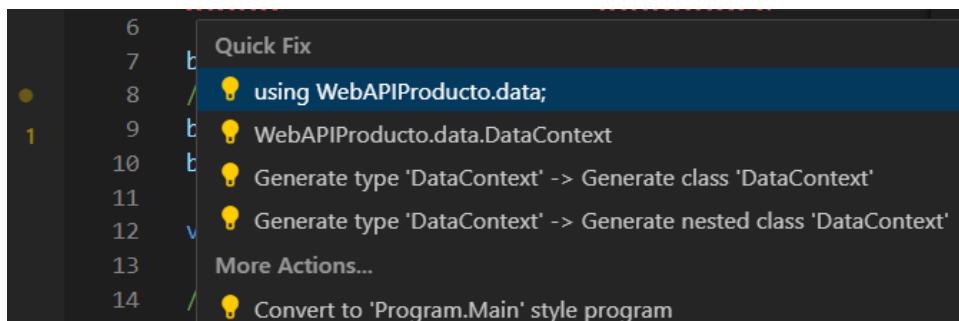
```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto> dotnet add package Microsoft.EntityFrameworkCore.Design
```

Ya añadidos los paquetes.



```
 1 <ItemGroup>
 2   <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="7.0.5" />
 3   <PackageReference Include="Microsoft.EntityFrameworkCore" Version="7.0.11" />
 4   <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="7.0.11">
 5     <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
 6     <PrivateAssets>all</PrivateAssets>
 7   </PackageReference>
 8   <PackageReference Include="Microsoft.EntityFrameworkCore.SQLite" Version="7.0.11" />
 9   <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
10 </ItemGroup>
11 </Project>
```

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 puntos
//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 parámetro (Es la clase)
    //option variable
    //base(options)pasa la variable a la clase base
    //La clase base es la que va a 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 dependencias 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
1  {
2      "Logging": {
3          "LogLevel": [
4              "Default": "Information",
5              "Microsoft.AspNetCore": "Warning"
6          ]
7      },
8      "AllowedHosts": "*",
9      "ConnectionStrings": {
10         "DefaultConnection": "Data Source=DataApi.db"
11     }
12 }
13 }
```

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

```
1  using Microsoft.EntityFrameworkCore;
2  using WebAPIProducto.data;
3
4  var builder = WebApplication.CreateBuilder(args); //constructor
5
6  // Add services to the container.
7  //para agregar las independencias para la conexión
8  //en UseSqlite() le entra la cadena de conexión a la BD
9  var cadenaConexion = builder.Configuration.GetConnectionString("DefaultConnection");
10
11 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 existe).

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

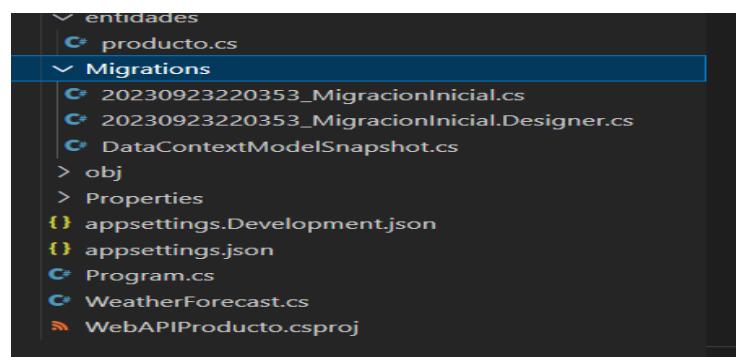
```
PS C:\Users\anaic\OneDrive\Escritorio\ApiNet\WebAPIProducto> dotnet tool install --global dotnet-ef
```

```
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



```
WebAPIProducto > Migrations > 20230923220353_MigracionInicial.cs > ...
1  using System;
2  using Microsoft.EntityFrameworkCore.Migrations;
3
4  #nullable disable
5
6  namespace WebAPIProducto.Migrations
7  {
8      /// <inheritdoc />
9      public partial class MigracionInicial : Migration
10     {
```

```

    /// <inheritdoc />
    protected override void Up(MigrationBuilder migrationBuilder) //método que va a escribir o hacer cambios en la BD
    {
        migrationBuilder.CreateTable<Producto>()
            .Name("Producto")
            .Columns(table => new
            {
                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
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Producto");
    }
}

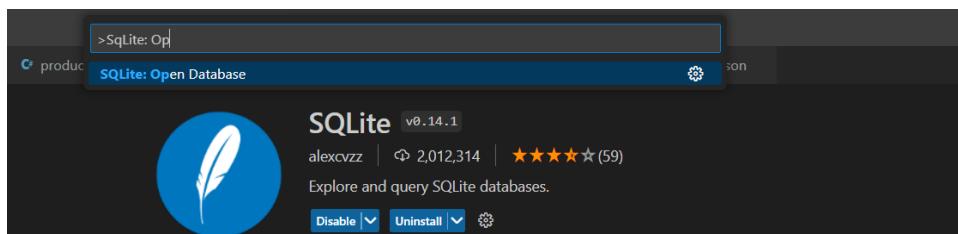
```

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.

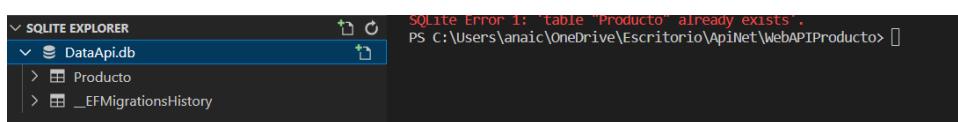
Ahora con la extensión SQLite.



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



Se puede ver la base de datos en SSQLITE EXPLORER



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

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

```

1  using Microsoft.AspNetCore.Mvc;
2  using Microsoft.EntityFrameworkCore;
3  using WebAPIProducto.data;
4  using WebAPIProducto.entidades;
5
6  namespace WebAPIProducto.Controllers
7  {
8      [ApiController]
9      [Route("api/[controller]")]
10     3 references
11     public class ProductoController : ControllerBase
12     {
13         //para registrar los eventos
14         1 reference
15         private readonly ILogger<ProductoController> _logger;
16         //Contexto
17         3 references
18         private readonly DataContext _context;
19         //constructor
20         0 references
21         public ProductoController(ILogger<ProductoController> logger, DataContext context )
22         {
23             _logger=logger;
24             _context=context;
25         }
26     }

```

```

//creación de método getAll (Los devuelve todos los registros)
[HttpGet(Name = "GetProductos")]
//traer lo productos de tipo asíncrona (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
```

The screenshot shows a Swagger UI interface for a 'Producto' API. The interface has a header 'Producto' and a list of five endpoints:

- GET /api/Producto (blue button)
- POST /api/Producto (green button, highlighted)
- PUT /api/Producto (orange button)
- GET /api/Producto/{id} (blue button)
- DELETE /api/Producto/{id} (red button)

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