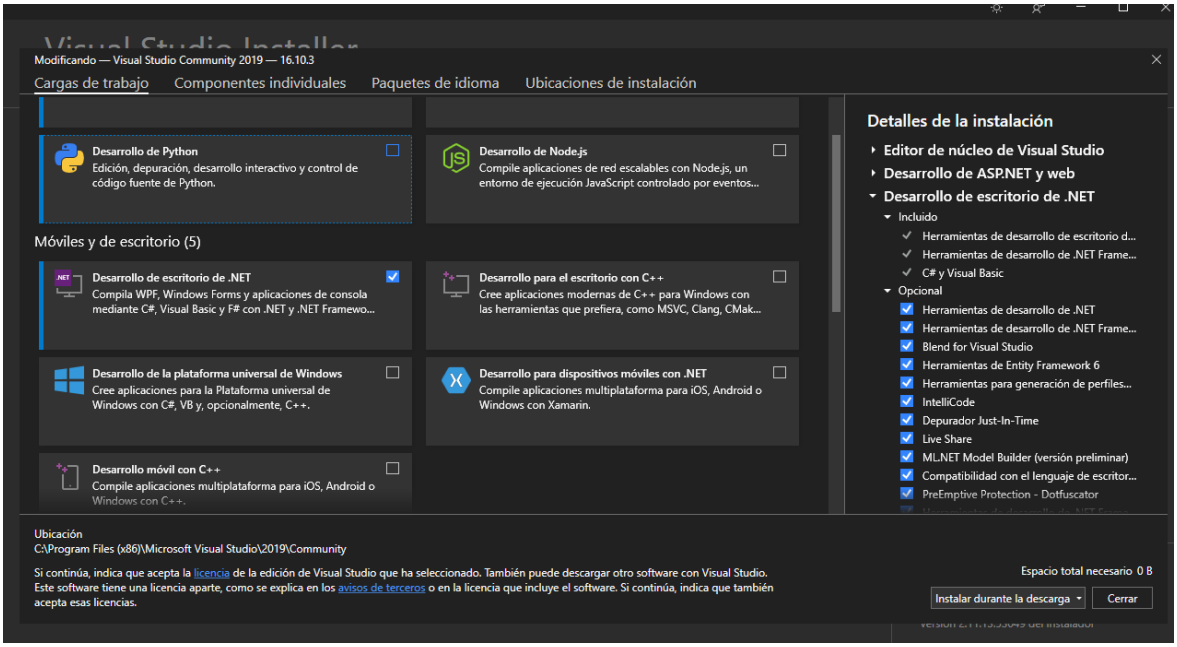
Práctica Microservicios y Contenedores

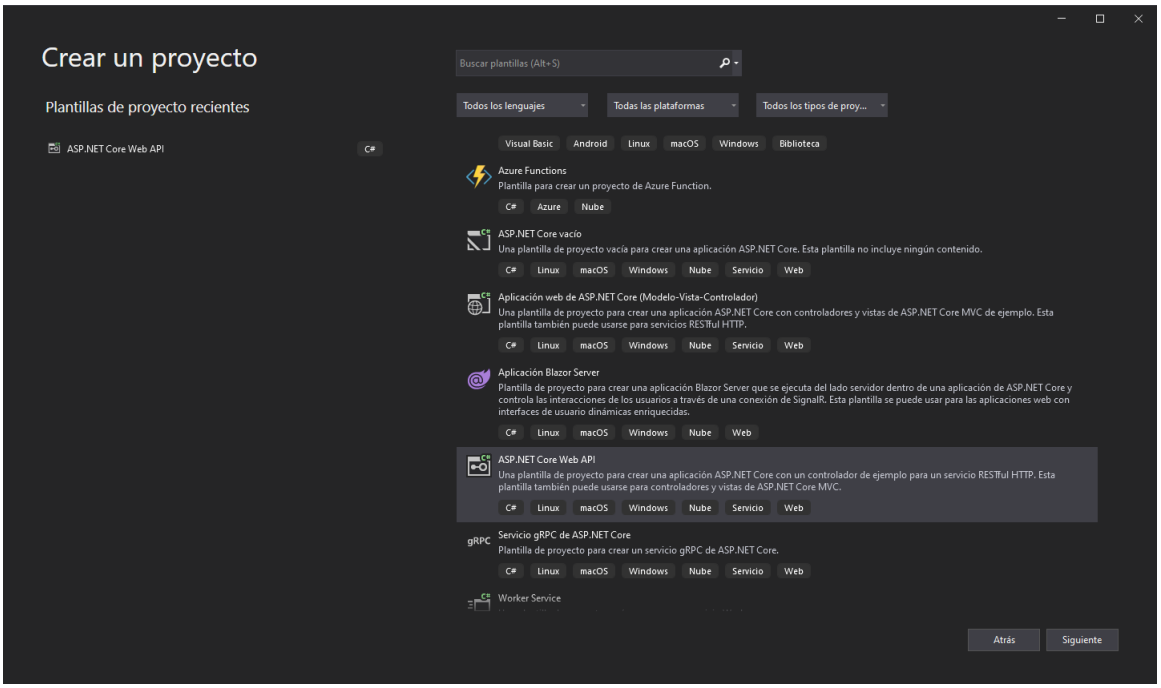
Se optó por desarrollar una Api en Visual Studio 2019 Net Core con el cual obtendremos, agregaremos, modificaremos y eliminaremos autores de libros de una base de datos SQL.

Creación del microservicio:

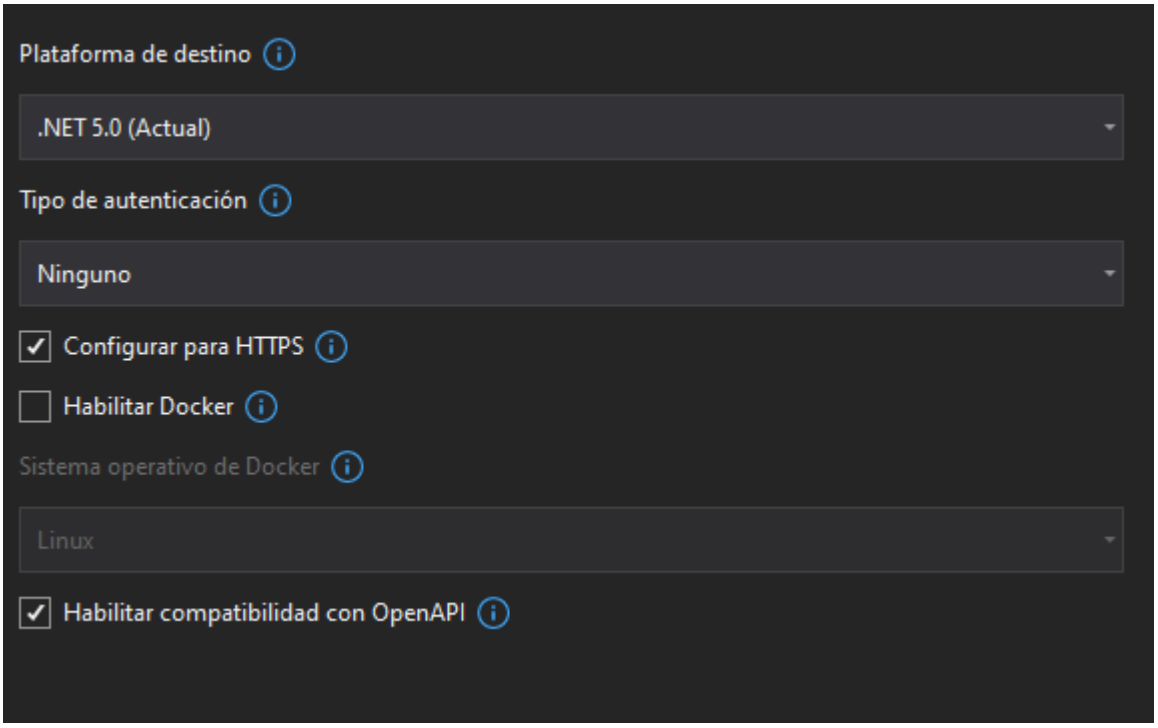
Para empezar, es necesario agregar a nuestro visual estudio las siguientes configuraciones:



Esto es necesario para que nuestro VisualStudio nos permita crear un proyecto ASP.NET Core Web API.



Después de definir como se llamará y donde se encontrará nuestro proyecto debemos agregar la siguiente configuración para poder trabajar de la forma mas eficiente. (Es necesario habilitar la compatibilidad con OpenAPI para que al levantar el micro podamos contar con una interfaz Swagger que hará más intuitiva las rutas que podemos utilizar).



Una vez creado nuestro proyecto debemos borrar el controlador crear las siguientes carpetas con las cuales les daremos una arquitectura correcta a nuestra API:

- DTO
- Entidades
- Utilidades

Ademas de agregar los siguientes Paquetes NuGet:

- **AutoMapper.Extensions.Microsoft.DependencyInjection.** (version 8.1.1): Este paquete será el encargado de permitirnos mapear los valores de los json's de entrada a los valores de nuestra base de datos.
- **Microsoft.EntityFrameworkCore.SqlServer.** (versión 6.0.0): Este paquete nos permitirá conectarnos a una base de datos Sql
- **Microsoft.EntityFrameworkCore.Tools.** (versión 6.0.0): Este paquete es complemento al Sql para poder realizar acciones sobre nuestra base de datos.
- **Swashbuckle.AspNetCore.** (versión 5.6.3): Este paquete agrega la capacidad de que al desplegar nuestro micro podamos verlo en una interfaz Swagger

Programando nuestra Api:

Inicializando nuestra Base de datos:

Para configurar nuestra base de datos primero en el appsettings agregamos el string para establecer la conexión:

```
"ConnectionStrings": {
  "defaultConnection": "Data Source=(localdb)\\mssqllocaldb;Initial Catalog=Practica;Integrated Security=True"
}
```

Despues en el startup agregamos la conexión a la base de datos:

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("defaultConnection")));
```

Despues en la carpeta de Entidades definimos el formato que tendran nuestras tablas en nuestro caso seran dos: *Autor y Libro:*

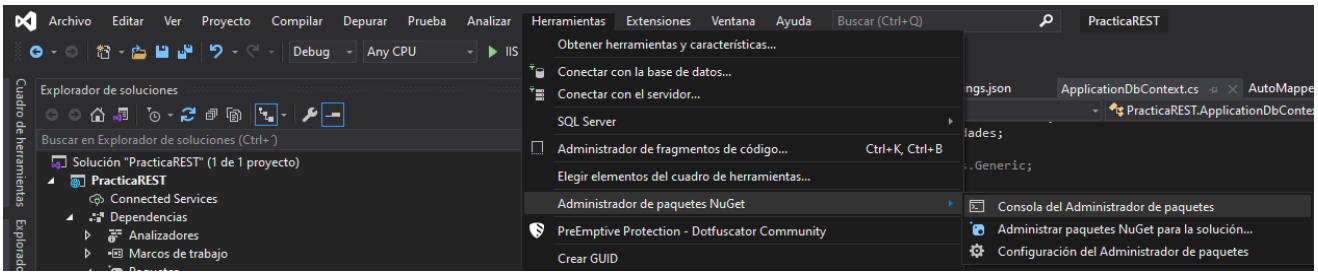
```
/// <summary>
/// Tabla Autor de la base de datos
/// </summary>
7 referencias
public class Autor
{
    5 referencias
    public int Id { get; set; }

    1 referencia
    public string Nombre{ get; set; }
```

```
public class Libro
{
    1 referencia
    public int Id { get; set; }
    0 referencias
    public string Titulo { get; set; }
    2 referencias
    public int AutorId { get; set; }

    1 referencia
    public Autor Autor { get; set; }

}
```

Despues agregamos en el proyecto agregamos una clase donde se agrega el set inicial para que al iniciar la migracion se creen las tablas con el formato que deseamos:

```
public class ApplicationDbContext:DbContext
{
    0 referencias
    public ApplicationDbContext(DbContextOptions options) : base(options)
    {

    }
    5 referencias
    public DbSet<Autor> Autores { get; set; }
    1 referencia
    public DbSet<Libro> Libros { get; set; }
}
```

Ahora para crear nuestra base de datos vamos a la pestaña de Herramientas, Administrador de paquetes NuGet, Consola de Administrador de Paquetes:



En la terminal, utilizamos el comando *Add-Migration Inicial* para crear nuestras tablas dentro de nuestra base de datos y Update-database pára terminar y poder empezar a usar nuestra base de datos

```
PM> Add-Migration Inicial
Build started...
Build succeeded.
```

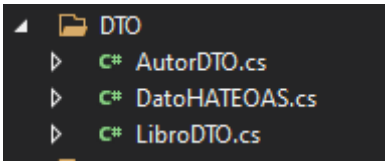
```
PM> Update-database
Build started...
Build succeeded.
Done.
PM>
```

Agregar AutoMapper:

Para empezar dentro dela configuracion de servicios de nuestro Startup, agregamos el AutoMapper de la siguiente forma:

```
services.AddAutoMapper(typeof(Startup));
```

Para poder agregar nuestros json de entrada, primero definimos el formato que tendran. Agregamos 2 Clases dentro de la carpeta DTO a los cuales les pondremos, *AutorDTO y LibroDTO:*



Despues (Deben coincidir con los valores de la tabla) agregamos los parametros que incluiran nuestros json, ademas de agregar las restriciones que creamos correctas, ejemplo definer que parametros son necesarios o el numero maximo de caracteres conendran:

```
public class AutorDTO
{
    [Required]
    [StringLength(maximumLength: 50, ErrorMessage = "El campo {0} no debe de tener mas de 50 caracteres")]
    2 referencias
    public string Nombre { get; set; }
}
```

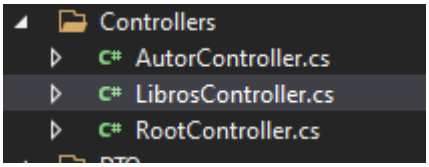
```
public class LibroDTO
{
    [Required]
    [StringLength(maximumLength: 50, ErrorMessage = "El campo {0} no debe de tener mas de 50 caracteres")]
    0 referencias
    public string Titulo { get; set; }
    0 referencias
    public int AutorId { get; set; }
}
```

Una vez definido el formato de nuestros jsons, dentro de la carpeta Utilidades Agregamos los perfiles que nuestro mapeo automatico realizara:

```
namespace PracticaREST.Utilidades
{
    /// <summary>
    /// Creacion de perfiles de mapeo
    /// </summary>
    1 referencia
    public class AutoMapperProfiles: Profile
    {
        0 referencias
        public AutoMapperProfiles()
        {
            CreateMap<AutorDTO, Autor>();
            CreateMap<LibroDTO, Libro>();
        }
    }
}
```

Agregar Controladores :

Dentro de la Carpeta Controller agregamos 3 clases con los siguientes nombres: *AutorController, LibrosController y RootController.*



En los controladores de Autor y Libro Agregamos la configuracion de context y mapper:

```
private readonly ApplicationDbContext context;
private readonly IMapper mapper;
/// <summary>
/// Método con el que agregamos la configuracion de los servicios
/// </summary>
/// <param name="context">interecion con la base de Datos </param>
/// <param name="mapper">configuracion del mapero</param>
0 referencias
public AutorController(ApplicationDbContext context, IMapper mapper)
{
    this.context = context;
    this.mapper = mapper;
}
```

Para agregar el HATEOAS, creamos una clase dentro de la carpeta DTO donde agregaremos el formato que tendran nuestras rutas en el controller Root:

```
public class DatoHATEOAS
{
    1 referencia
    public string Enlace { get; private set; }
    1 referencia
    public string Descripcion { get; private set; }
    1 referencia
    public string Metodo { get; private set; }
    /// <summary>
    /// Método que mapea los datos a nuestro
    /// </summary>
    /// <param name="enlace">ruta donde se encuentra</param>
    /// <param name="descripcion">Descripcion de que hace</param>
    /// <param name="metodo">que método REST es</param>
    6 referencias
    public DatoHATEOAS(string enlace, string descripcion, string metodo)
    {
        Enlace = enlace;
        Descripcion = descripcion;
        Metodo = metodo;
    }
}
```

Despúes definimos que haran nuestros métodos acontinuacion se muestra la salida de estos métodos cuando se despliega en nuestra maquina:

RootController

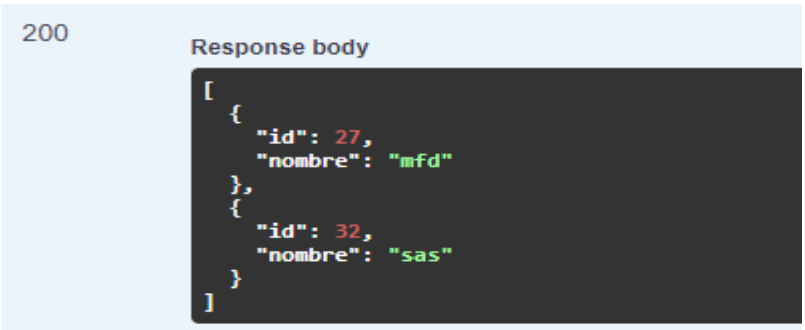
- Método Obtener Rutas: https://localhost:44321/api


```
[
  {
    "enlace": "https://localhost:44321/api",
    "descripcion": "self",
    "metodo": "GET"
  },
  {
    "enlace": "https://localhost:44321/api/autores",
    "descripcion": "autores",
    "metodo": "GET"
  },
  {
```

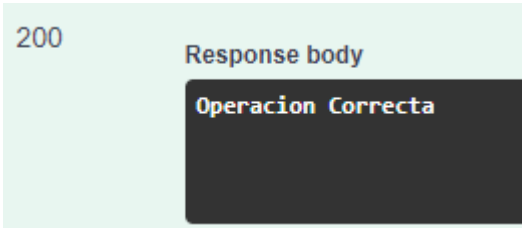
```
    "enlace": "https://localhost:44321/api/autores",
    "descripcion": "autor-crear",
    "metodo": "POST"
  },
  {
    "enlace": "https://localhost:44321/api/autores/Put",
    "descripcion": "Actualizar-autor-por-id-FromQuery",
    "metodo": "PUT"
  },
  {
    "enlace": "https://localhost:44321/api/autores/delete",
    "descripcion": "Borrar-autor-por-id-FromQuery",
    "metodo": "DELETE"
  },
  {
    "enlace": "https://localhost:44321/api/libros",
    "descripcion": "libro-crear",
    "metodo": "POST"
  }
]
```


AutorController

- Método Obtener Autores: https://localhost:44321/api/autores:



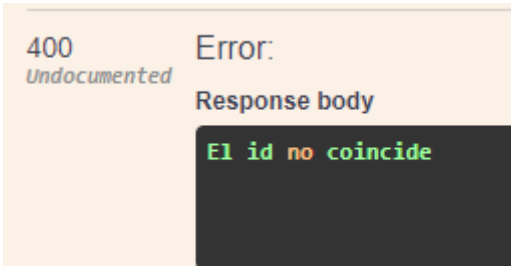
- Método Agregar Autor: https://localhost:44321/api/autores:
  - Request:



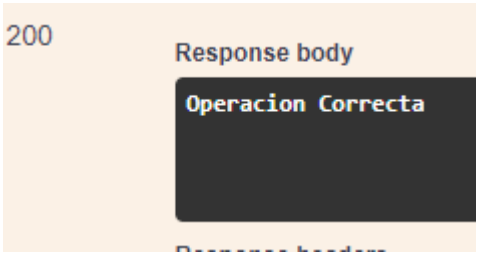
- Método Modificar Autor: https://localhost:44321/api/autores/Put?id=27:
  - Request:

```
{
  "id": int,
  "nombre": "string"
}
```

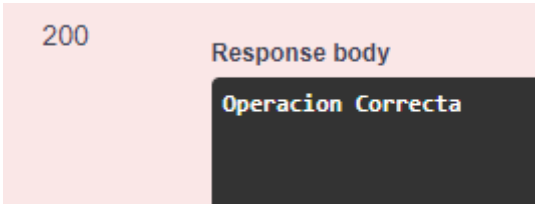
Respuesta Error:



Respuesta Correcta:



- Método Borrar Autor: https://localhost:44321/api/autores/delete?id=int2



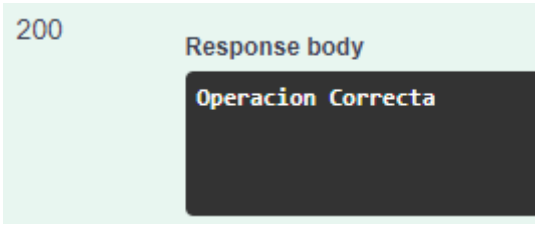
LibrosController

- Método Obtener Libros: https://localhost:44321/api/libros/int

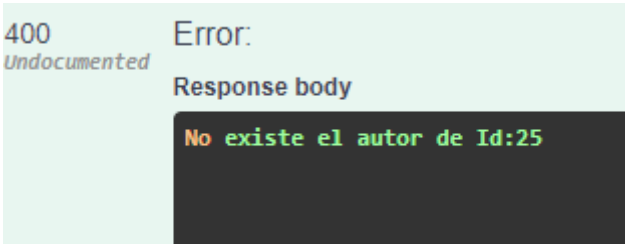
- Método Agregar Libro: https://localhost:44321/api/libros:
  - Request:

```
{
  "titulo": "string",
  "autorId": int
}
```

Respuesta Correcta:

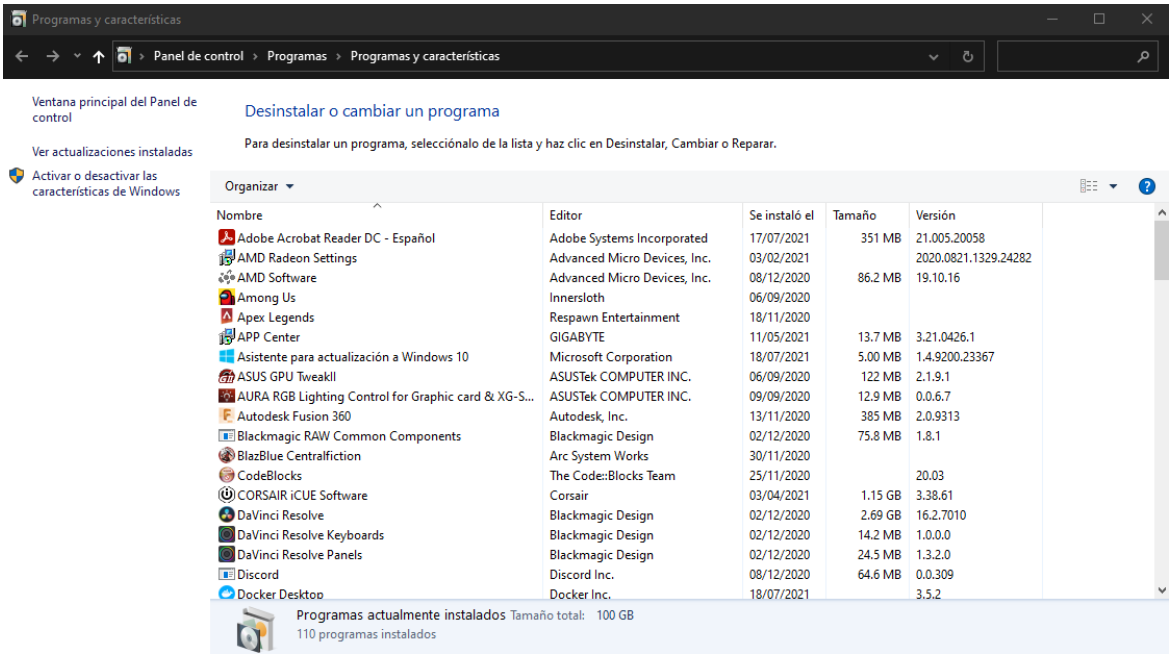


Error:

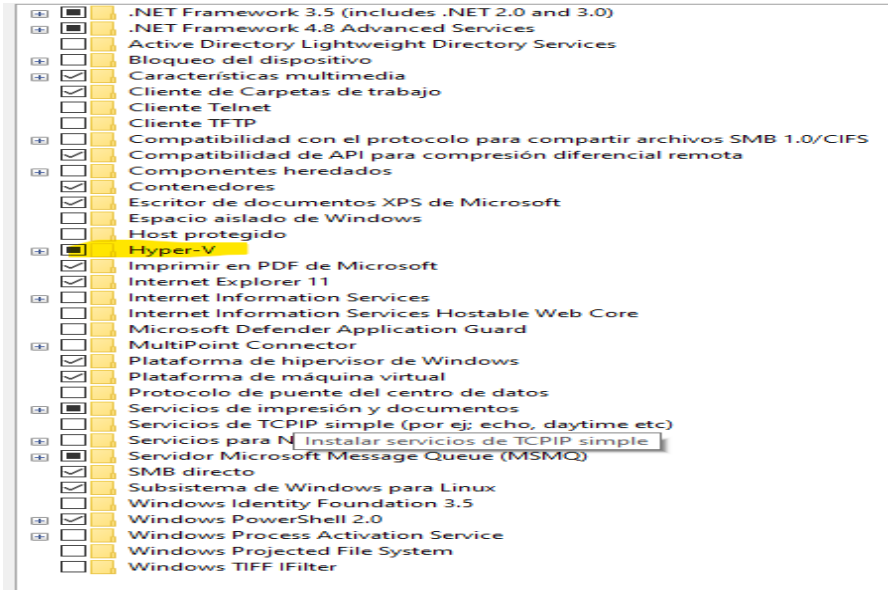


Para contenerizar la aplicación primero debemos tener instalado en nuestra maquina Docker, para esto es necesario primero tener una version Pro o Enterprise de Windows 10. Ademas de tener habilitado Hyper-V en nuestro windows:

Para esto primero debemos ir a Programas y caracteristicas y damos click en Activar o desactivar las caracteristicas de Windows:

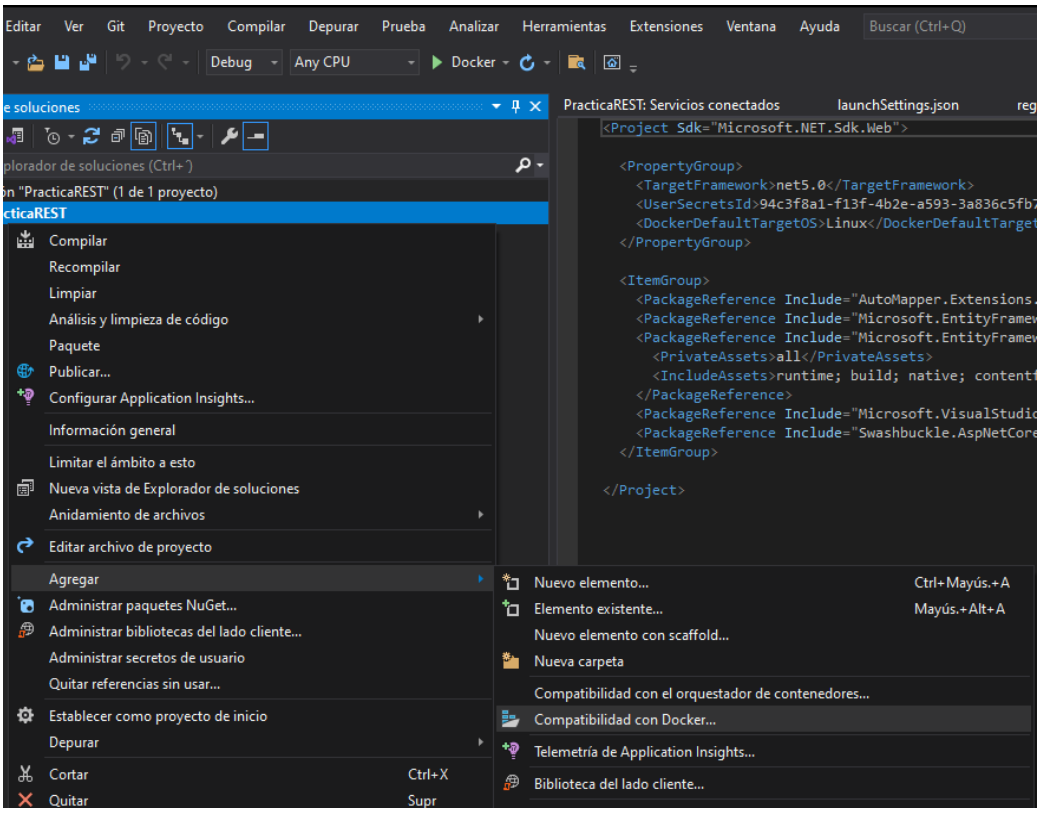


Despues damos click en Hyper-V

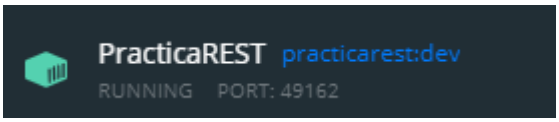


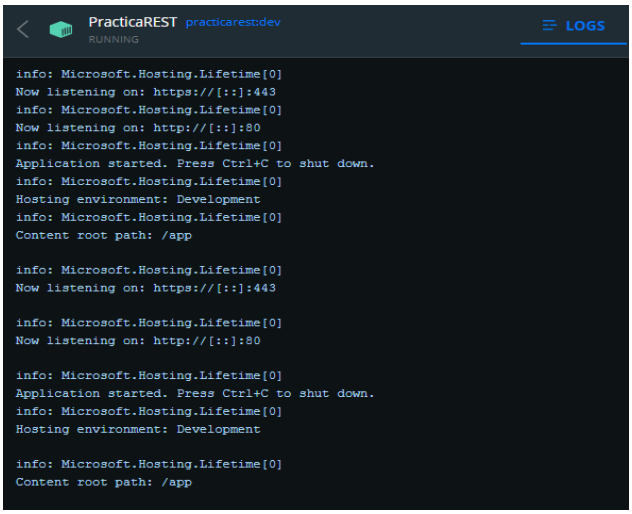
Si al instalarlo tienes problemas debemos revisar la BIOS para ver si esta habilitado la Virtualizacion.

Para Contenertizar la applicacion debemos dar click derecho sobre nuestra API, luego en agregar, y añadimos la Compativilidad con Docker.

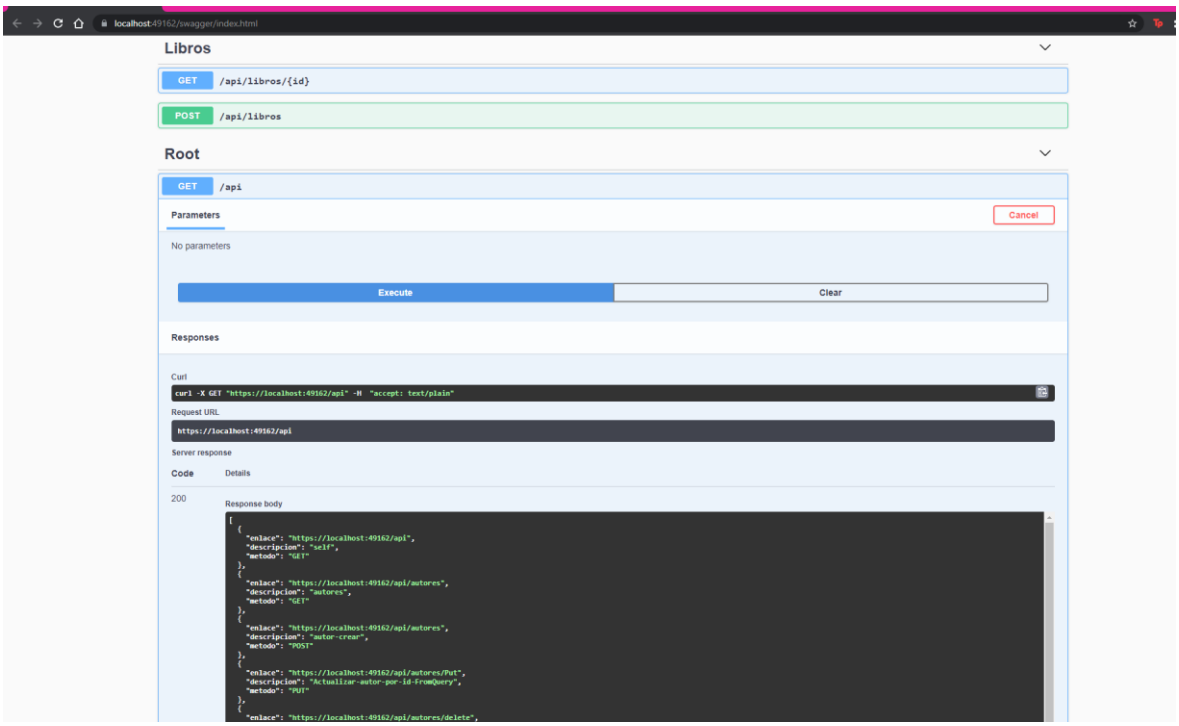


Todo esto nos permititira levantar el contenedor y podremos ver nuestra Api corriendo. Al agregar la compativilidad a nuestra aplicación al correr la aplicación autamaticamente levantara el contenedor dentro de Docker:



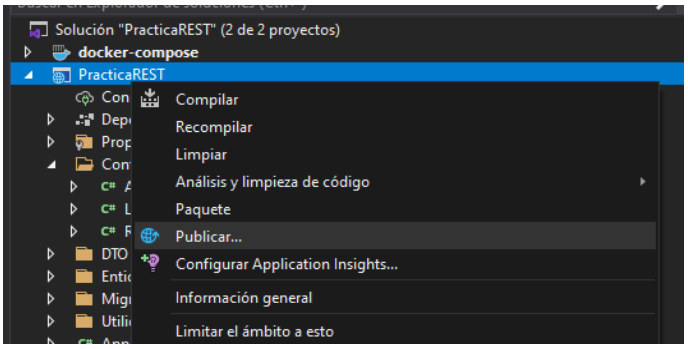


Al entrar a la url podemos ver: https://localhost:49162/swagger/index.html

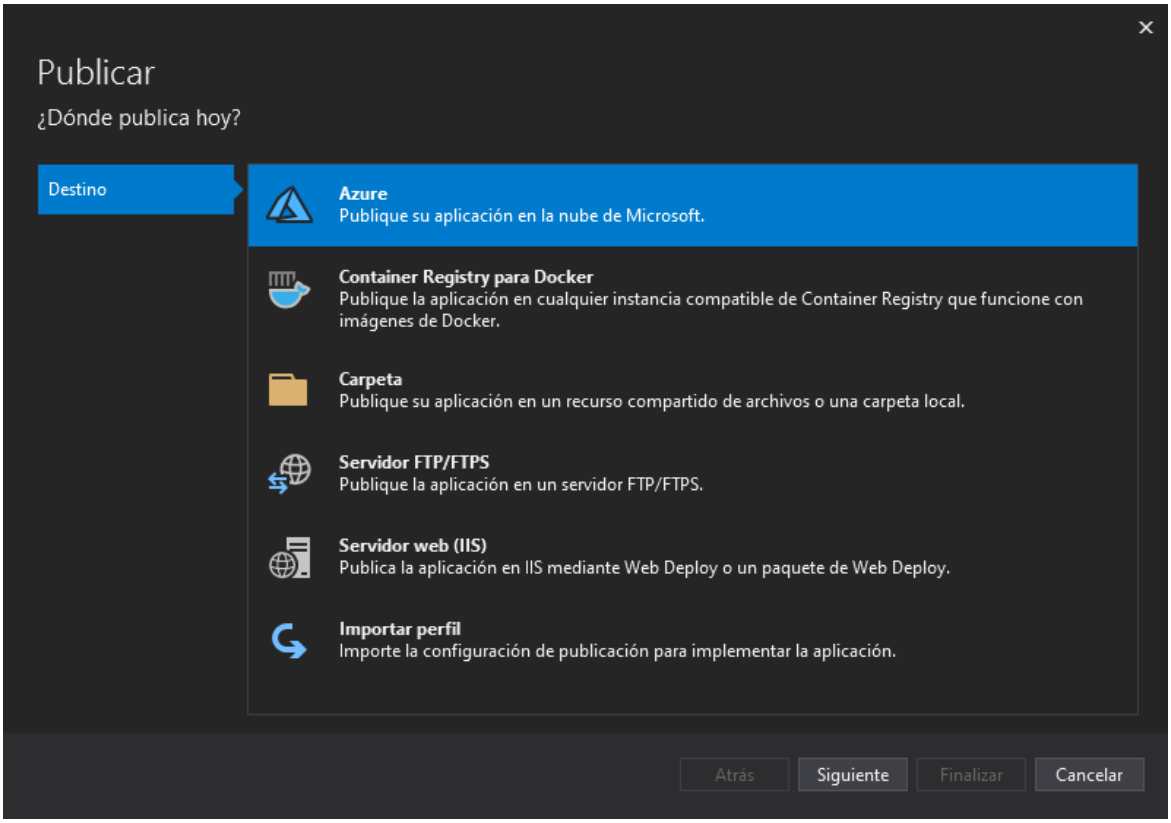


Desplegando Servicio:

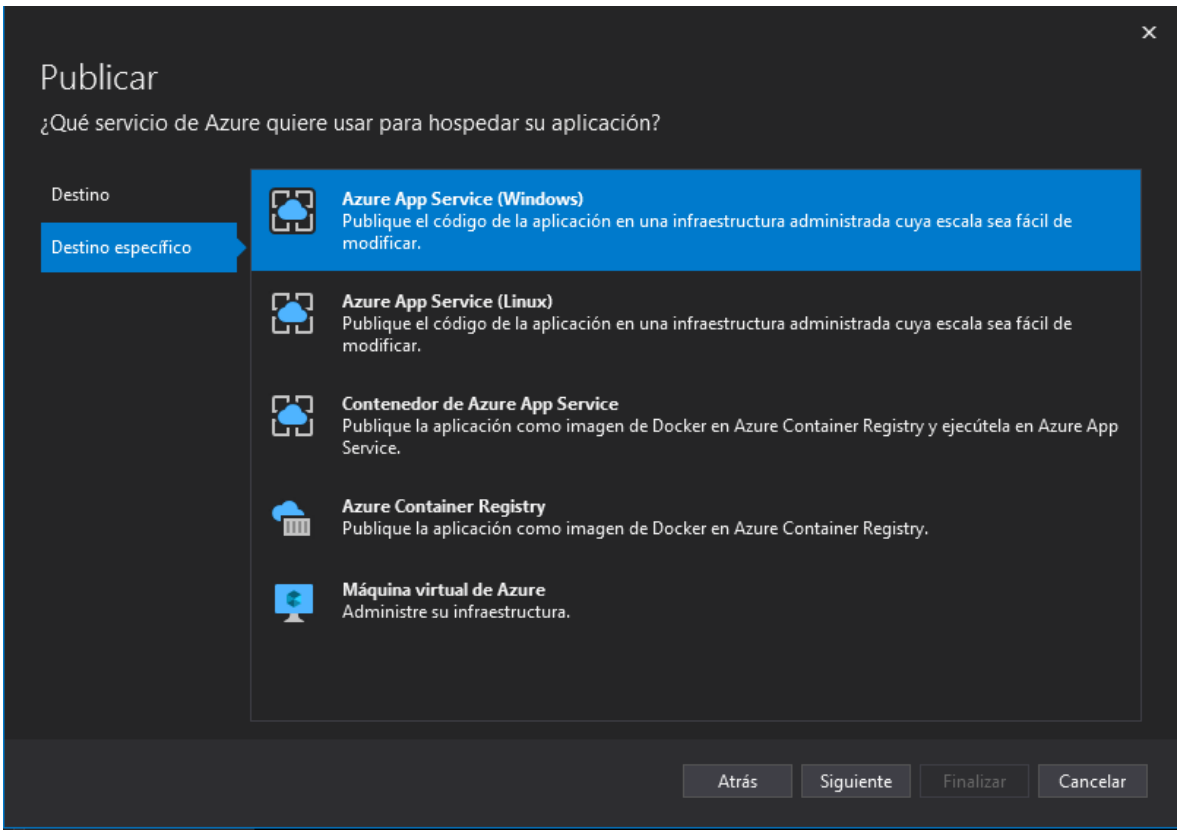
Para desdplegar nuestro servicio se obto por utilizar Azure, es necesario contar con cuenta para poder utilizar esta nube, para publicar algo dentro de VisualStudio 2019 damos click derecho al proyecto y selecionamos publicar:



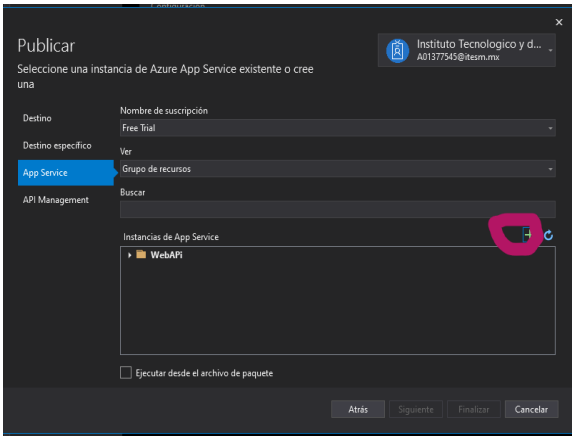
En publicar selecionamos Azure y damos click en Siguiente:



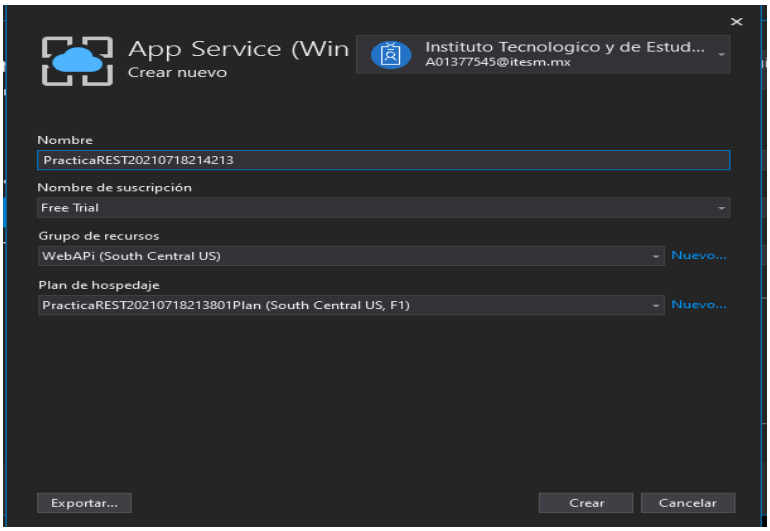
Como Destino especifico selecionamos Azure App Service(Windows) y damos click en siguientes:



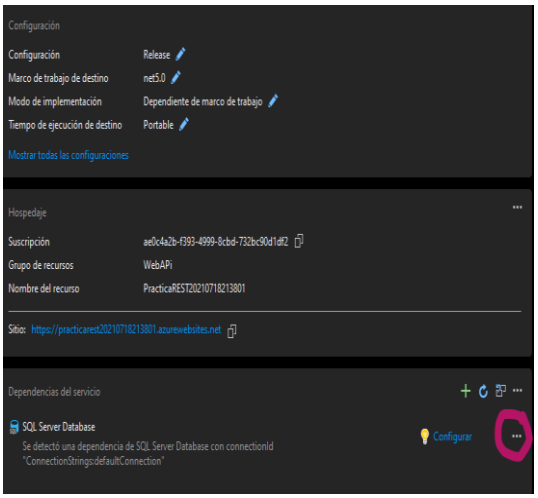
Dentro del App Service creamos una nueva Instancia:

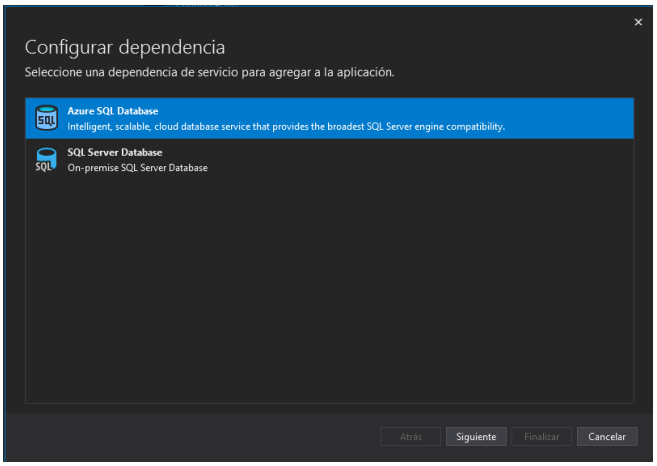


Dependiendo la capacidad del servidor sera el costo, en nuestro caso utilizaremos la version gratuita por que la Api no requiere de mucha capacidad.

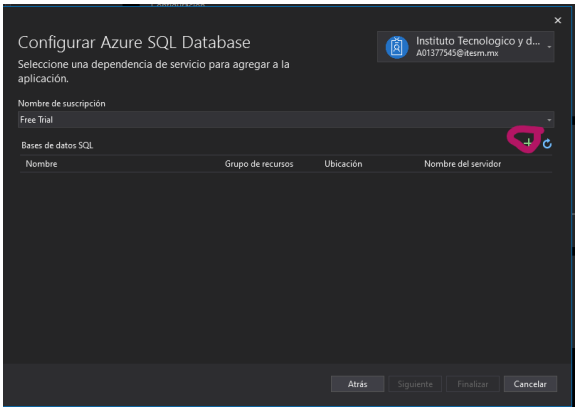


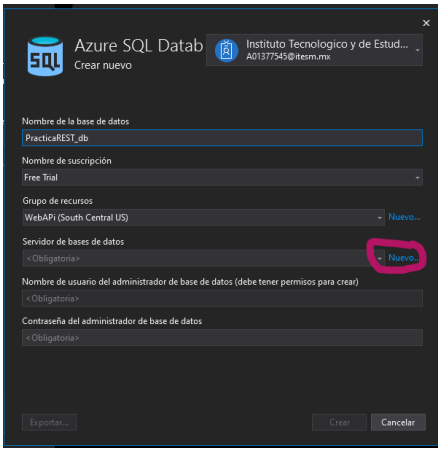
Antes de publicar nuestro proyecto es necesario crear una base de datos en Azure para poder utilizar nuestra API.





Damos click en + para crear una nueva base de datos, y creamos un nuevo servidor SQL:

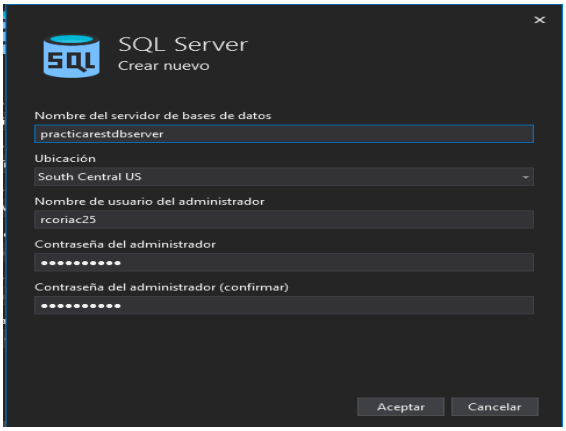




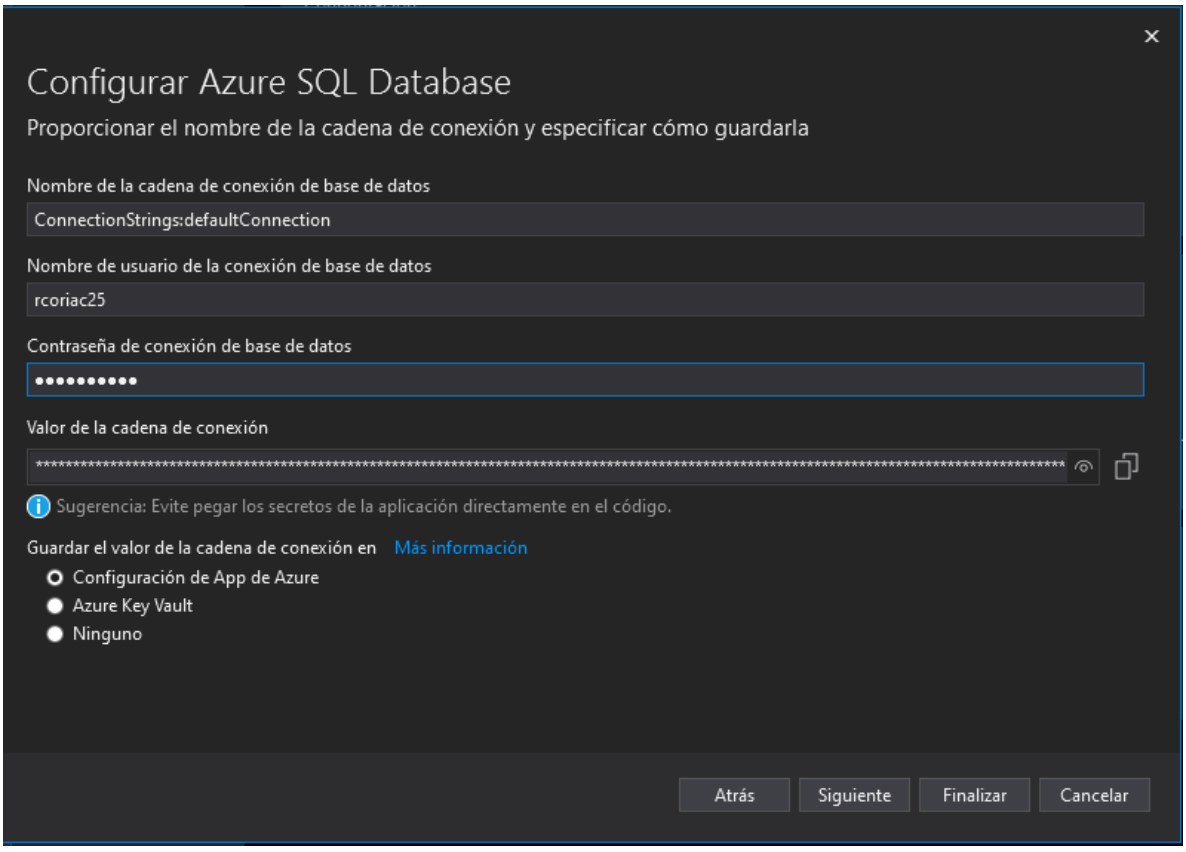
Para crear nuestro servidor debemos agregar un nombre, un usuario y un password en este caso:

Usuario:rcoriac25

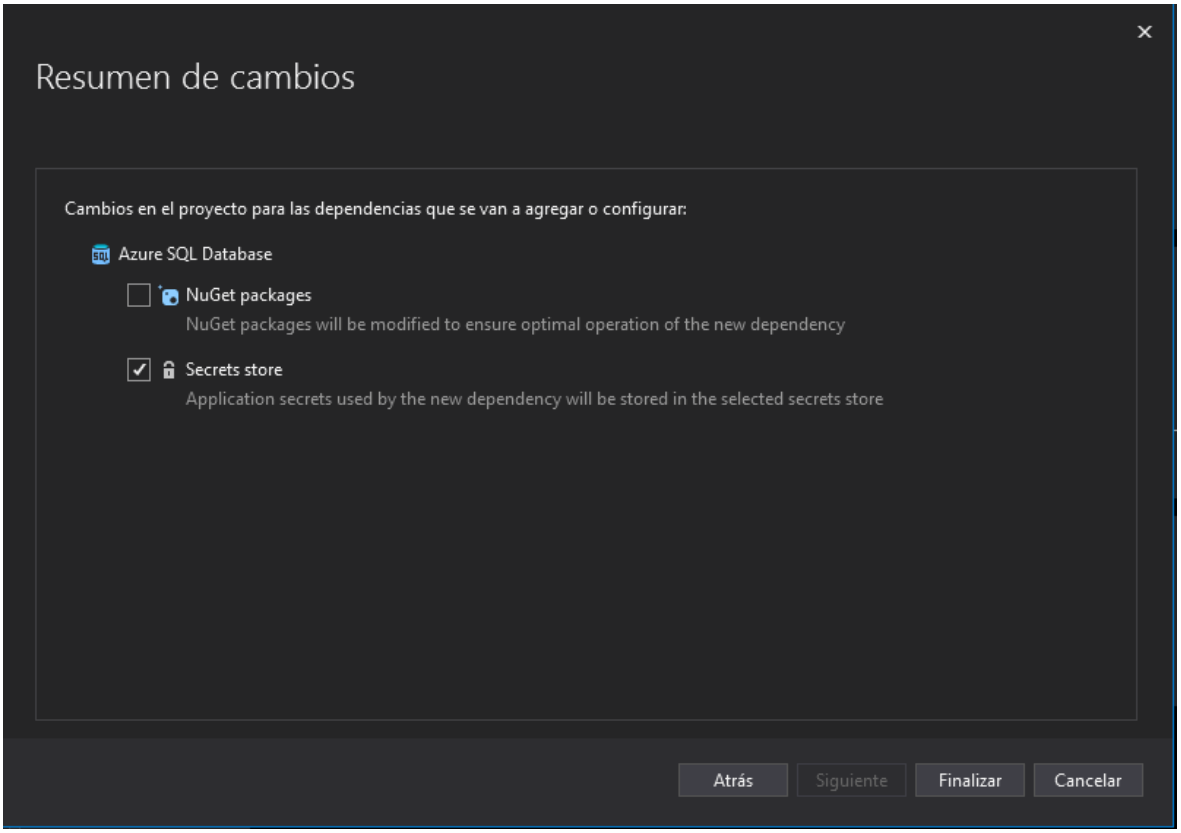
Password:admin5050/



Despues debemos colocar como nombre de cadena defaultConnection dado que asi lo configuramos en el Startup

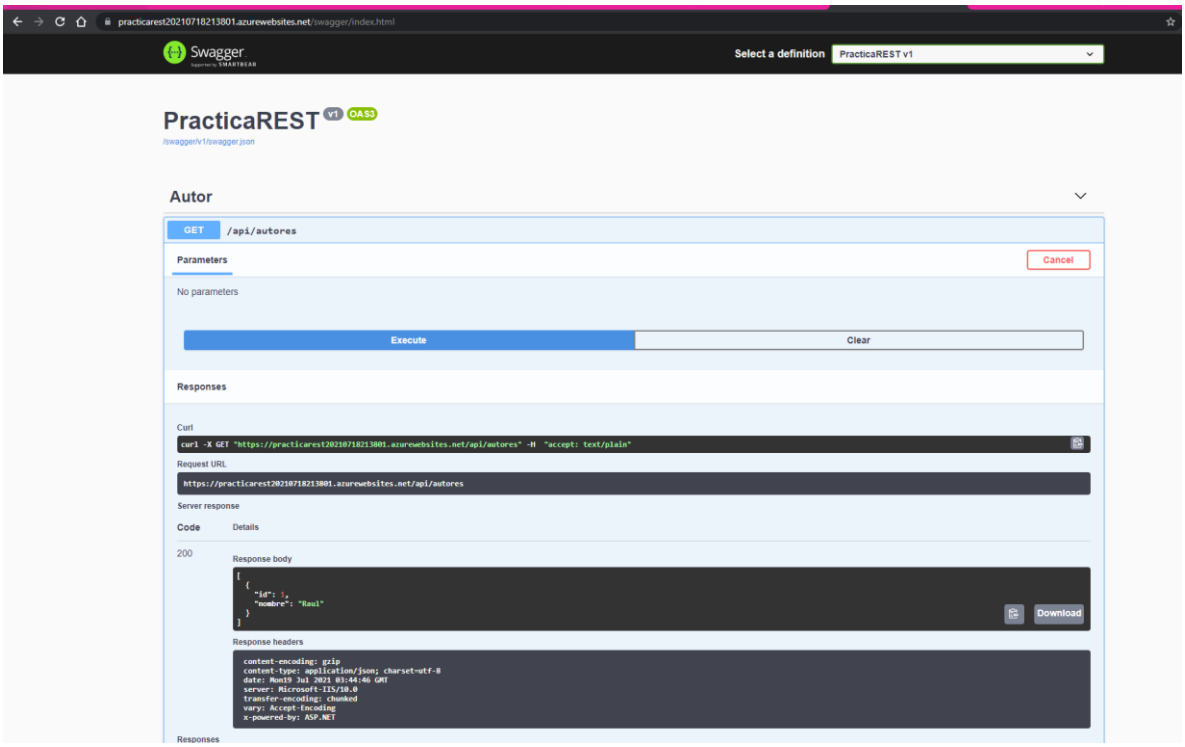


Para finalizar deselecionamos que se agreguen los NuGet packages:



Al dar click en publicar nuestra API estara desplegada en la siguiente liga:

https://practicarest20210718213801.azurewebsites.net/swagger/index.html


practicarest20210718213801.azurewebsites.net/swagger/index.html
Swagger
Select a definition
PracticaREST v1
PracticaREST
v1
OAS3
/swagger/v1/swagger.json
Autor
GET
/api/autores
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl
curl -X GET "https://practicarest20210718213801.azurewebsites.net/api/autores" -H "accept: text/plain"
Request URL
https://practicarest20210718213801.azurewebsites.net/api/autores
Server response
Code
Details
200
Response body
[
{
"id": 1,
"nombre": "Raul"
}
]
Download
Response headers
content-encoding: gzip
content-type: application/json; charset=utf-8
date: Mon19 Jul 2021 03:44:46 GMT
server: Microsoft-IIS/10.0
transfer-encoding: chunked
vary: Accept-Encoding
x-powered-by: ASP.NET
Responses