



# TI-Capital Humano

Desarrollador .Net

Aplicación Web MVC en 3  
Capas



## Contenido

1.	Introducción.....	4
2.	Crear Arquitectura de la Aplicación.....	4
2.1.	Crear Solución Vacía .....	4
2.2.	Crear las Capas .....	4
2.2.1.	Agregar Proyectos de tipo Biblioteca de Clases.....	4
2.2.2.	Agregar Proyectos de tipo MVC .....	5
2.3.	Agregar Referencias entre Proyectos .....	5
3.	Crear Capa de Datos .....	5
3.1.	Agregar Paquetes Nugets.....	6
3.2.	Mapear la Base de datos con EntityFrameWork.....	6
3.3.	Crear la Interface IRepository .....	8
3.4.	Crear la Clase que utiliza EntityFramework .....	9
4.	Crear Capa de Negocios.....	11
4.1.	Agregar Paquetes NuGet.....	11
4.2.	Crear la Clase que utiliza EntityFramework .....	11
4.3.	Crear la Clase que consume la Web API.....	12
5.	Crear Capa de Presentación.....	15
5.1.	Agregar la URL en appsettings.json .....	15
5.2.	Agregar el Controlador.....	16
5.2.1.	Definir Variables Globales .....	16
5.2.2.	Crear el constructor .....	16
5.2.3.	Cambiar el Métodos para que sean asíncronos .....	17
5.2.1.	Crear el método privado Consultar.....	17
5.2.2.	Modificar el método Index .....	18
5.2.3.	Modificar el método Details .....	18
5.2.4.	Modificar el método Delete.....	18
5.2.5.	Modificar el método Create .....	18
5.2.6.	Modificar el método Edit.....	19
5.3.	Registrar los Servicios para la Inyección de Dependencias.....	20
5.4.	Agregar las Vistas.....	21
5.4.1.	Vista Index.....	21
5.4.2.	Vista Details .....	21
5.4.3.	Vista Create.....	22
5.4.4.	Vista Edit .....	23

5.4.5.	Vista Delete .....	24
5.5.	Ejecutar la Aplicación Web MVC .....	24
5.5.1.	Establecer Proyecto de Inicio .....	24
5.5.2.	Cambiar el Enrutamiento por Default .....	24
5.5.3.	Ejecutar la Aplicación .....	25
5.5.4.	Cambiar la clase de Negocio.....	25
5.5.5.	Ejecutar la Aplicación consumiendo la API .....	26

# 1. Introducción

El objetivo de esta guía es proporcionar los pasos generales para crear una aplicación web CRUD para la entidad **Estados**, utilizando ASP .Net Core 6. La arquitectura será de tres capas, con un proyecto MVC en la capa de presentación.

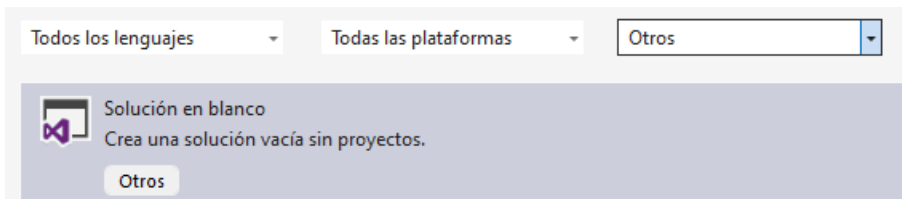
En la capa de datos se creará con EntityFramework.

Se utilizará el patrón de diseño Repository, asimismo en la capa de negocio se tendrán dos clases que implementen la interfaz del repositorio, una que accesa la base de datos directamente desde Entity Framework, y la otra consumiendo una Web API con los métodos CRUD implementados. La clase de negocio será inyectada en el controlador.

El controlador se creará con la Plantilla de Acciones de Lectura y Escritura, por lo que también se crearán las Vistas correspondientes.

## 2. Crear Arquitectura de la Aplicación

### 2.1. Crear Solución Vacía

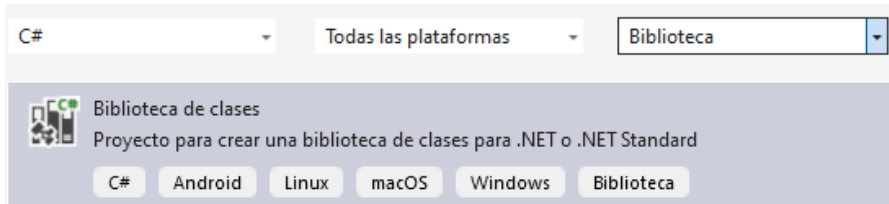


### 2.2. Crear las Capas

#### 2.2.1. Agregar Proyectos para Capa de Datos y Negocio, y Entidades

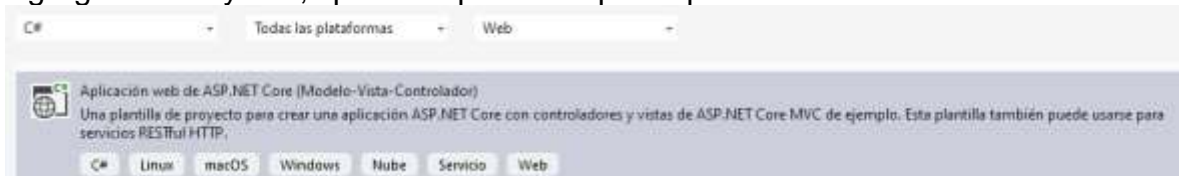
Agregar tres Proyectos de tipo Biblioteca de Clases, con framework .Net 6.0. Para las capas de Datos y Negocio, y el tercero es para desacoplar las clases tipo POCO en el proyecto Entidades:

- Entidades
- Datos
- Negocio



### 2.2.2. Agregar Proyectos para Capa de Presentación

Agregar un Proyecto, tipo MVC para la capa de presentación



Aplicación web de ASP.NET Core (Modelo-Vista-Controlador) C#

Framework ⓘ  
.NET 6.0 (Compatibilidad a largo plazo)

Authentication de campo ⓘ  
Ninguno

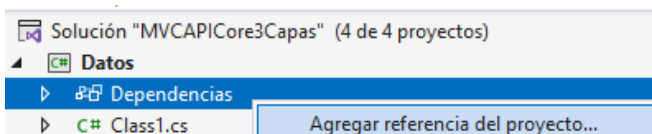
☐ Configurar para HTTPS ⓘ  
☐ Habilitar compatibilidad con el contenedor ⓘ

SO del contenedor ⓘ  
Linux

Tipo de compilación de contenedor ⓘ  
Dockerfile

☐ No usar instrucciones de nivel superior ⓘ

### 2.3. Agregar Referencias entre Proyectos



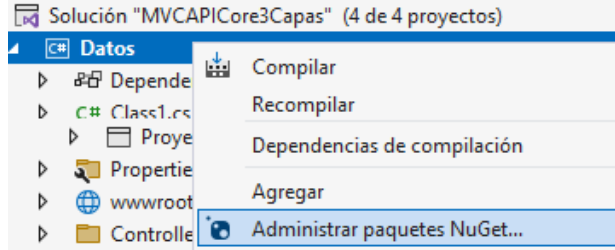
- El Proyecto Datos hace referencias al Proyecto Entidades
- El Proyecto Negocio hace referencias a los Proyectos Entidades y Datos
- El Proyecto Presentación hace referencias a los Proyectos Entidades y Negocio

## 3. Crear Capa de Datos

Para la creación de la capa de datos se utilizará el ORM EntityFramework

### 3.1. Agregar Paquetes Nugets

Agregar los siguientes paquetes, en el Proyecto de Datos:

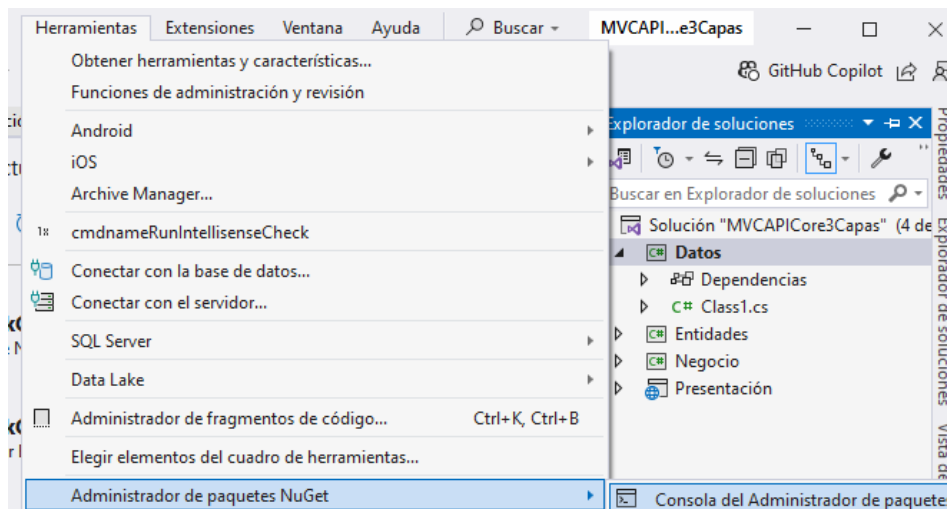


Microsoft.EntityFrameworkCore.SqlServer ver 6.0.32

Microsoft.EntityFrameworkCore.Tools ver 6.0.32

### 3.2. Mapear la Base de datos con EntityFramework

Estando posicionado en el Proyecto Datos, ejecutar en la consola del Administrador de Paquetes NuGet, el proceso de scaffolding para obtener las clases de tipo de entidad y una clase DbContext basada en un esquema de base de datos. Utilizar el comando Scaffold-DbContext en la consola del Administrador de paquetes (PMC) de EF Core



Recordar que scaffolding de ASP.NET es la generación de código preinstalados para proyectos de MVC y API web.

```
Scaffold-DbContext "Server=MSI;Database=InstitutoTich;User
ID=sa;password=Pass2017;" Microsoft.EntityFrameworkCore.SqlServer -Tables
"Estados" -Context EdoContext -ContextDir . -OutputDir ..\Entidades -NoPluralize -
Force
```

Utilizar la ruta relativa para indicar en que carpeta sedebe colocar la clase DbContext y la (o las) clases tipo POCO(Entidad). Teniendo en cuenta que el

directorio de trabajo es el Proyecto Datos. Por lo que en este caso se está solicitando que la clase DbContext se guarde en la carpeta del proyecto Datos y las entidades en la carpeta del Proyecto Entidades.

**-ContextDir . -OutputDir ..\Entidades**

```
PM> Scaffold-DbContext "Server=MSI;Database=InstitutoTich;User ID=sa;password=112j1ndr1;"
Microsoft.EntityFrameworkCore.SqlServer -Tables "Estados" -Context EdoContext -ContextDir . -OutputDir ..\Entidades -
NoPluralize -Force
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string, you should move it out of source code. You can avoid
scaffolding the connection string by using the Name= syntax to read it from configuration - see https://go.microsoft.com/
/fwlink/?linkid=2131148. For more guidance on storing connection strings, see http://go.microsoft.com/fwlink/?LinkId=723263.
PM>
```

Toda vez que se ha creado la clase de DbContext y la de Entidad

- Cambiar el NameSpace de las clases tipo POCO creadas en el Proyecto de Entidades.

```
namespace Datos
{
    2 referencias
    public partial class Estados
    {

```

➔

```
namespace Entidades
{
    0 referencias
    public partial class Estados
    {

```

- Agregar a la clase DbContext, el using Entidades
- Ajustar el método OnConfiguring de la clase de Contexto, ya que en ella el scaffolding deja las credenciales de la base de datos, mismas que deberán quedar en el archivo de configuración **appsettings.json**

```
public partial class EdoContext : DbContext
{
    0 referencias
    public EdoContext()
    {
    }

    0 referencias
    public EdoContext(DbContextOptions<EdoContext> options)
        : base(options)
    {
    }

    0 referencias
    public virtual DbSet<Estados> Estados { get; set; } = null!;

    0 referencias
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            warning To protect potentially sensitive information in your connection string, you should move it out of s
            optionsBuilder.UseSqlServer("Server=MSI;Database=InstitutoTich;User ID=sa;password=Pass2017;");
        }
    }
}
```

La cadena de conexión se pasará al archivo de configuración appconfig.json, del Proyecto Presentación.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "InstitutoTich": "Server=MSI;Database=InstitutoTich;User ID=sa;password=Pass2017;"
  }
}
```

En el archivo **program.cs**, del Proyecto de Presentación, se deberá cargar esta cadena de conexión como sigue:

```
using Microsoft.EntityFrameworkCore;
using CRUEstadosMVCCore.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<EdoContext>(opcion => {
    opcion.UseSqlServer(builder.Configuration.GetConnectionString("InstitutoTich"));
});

var app = builder.Build();
```

El `builder.Services.AddDbContext<EdoContext>` está registrando el objeto `EdoContext` (`DbContext`), para posteriormente inyectarlo en el objeto que dependa de él.

Agregar las referencias para la clase `UseSqlServer` y para la clase de contexto

El `builder.Configuration` obtendrá la cadena de conexión del archivo de configuración que se encuentre con el nombre proporcionado, en este caso `InstitutoTich`, y el `builder.Services.AddDbContext` se la pasa a la clase de contexto.

Ajustar el método `OnConfiguring` de la clase de Contexto, dejando el cuerpo vacío.

```
0 referencias
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
}
```

### 3.3. Crear la Interface IRepository



Con la finalidad de implementar el Patrón de Diseño Repositorio crear la Interface IRepository con métodos CRUD en el proyecto Entidades, e implementar dos clases:

```
public interface IRepository<TEntidad> where TEntidad : class
{
    2 referencias
    public Task<List<TEntidad>> Consultar();
    2 referencias
    public Task<TEntidad> Consultar(int? id);
    2 referencias
    public Task<TEntidad> Agregar(TEntidad entidad);
    2 referencias
    public Task Actualizar(TEntidad entidad);
    2 referencias
    public Task Eliminar(int id);
}
```

- 1) **DEstadoEF**.- Implementará los métodos utilizando DbContext de la capa de Datos
- 2) **NEstadoAPI**.- Implementará los métodos mediante el consumo de una Web API.

La Interface se define Genérica para que se pueda implementar para cualquier clase Entidad

### 3.4. Crear la Clase que utiliza EntityFramework

Crear la clase **DEstadoEF** que implemente la Interface **IRepository**, y en sus métodos utilizará **DbContext**.

En esta clase se creará el constructor que reciba por medio de inyección de dependencias, el objeto **DbContext**, creada con el scaffolding de EntityFramework, y se pueda tener acceso a la Base de datos.

```

public class DEstadoEF : IRepository<Estados>
{
    private readonly EdoContext _context;

    0 referencias
    public DEstadoEF(EdoContext context)
    {
        _context = context;
    }

    2 referencias
    public async Task Actualizar(Estados entidad)
    {
        _context.Update(entidad);
        await _context.SaveChangesAsync();
    }

    2 referencias
    public async Task<Estados> Agregar(Estados entidad)
    {
        _context.Add(entidad);
        await _context.SaveChangesAsync();
        return entidad;
    }

    2 referencias
    public async Task<List<Estados>> Consultar()
    {
        if (_context.Estados != null) return await _context.Estados.ToListAsync();
        throw new Exception($"Entity set 'EdoContext.Estados' is null.");
    }

    2 referencias
    public async Task<Estados> Consultar(int? id)
    {
        if (_context.Estados != null)
            return await _context.Estados.FirstOrDefaultAsync(m => m.Id == id);
        throw new Exception($"Entity set 'EdoContext.Estados' is null.");
    }

    2 referencias
    public async Task Eliminar(int id)
    {
        if (_context.Estados == null)
        {
            throw new Exception($"Entity set 'EdoContext.Estados' is null.");
        }
        var estados = await _context.Estados.FindAsync(id);
        if (estados != null)
        {
            _context.Estados.Remove(estados);
        }

        await _context.SaveChangesAsync();
    }
}
  
```

## 4. Crear Capa de Negocios

### 4.1. Agregar Paquetes NuGet

Agregar los siguientes paquetes que se necesitarán para leer el archivo de configuración y serializar y deserializar objetos.

- Microsoft.Extensions.Configuration 6.0
- Newtonsoft.Json 13.0.3

### 4.2. Crear la Clase que utiliza EntityFramework

Crear la clase **NEstadoEF** que utiliza la clase de la capa de Datos, la cual implementa la Interface **IRepositorio**, y en sus métodos utiliza **DbContext**

En esta clase se creará el constructor que reciba por medio de inyección de dependencias, el objeto **DEstadoEF**, para que es la que realmente utiliza el EntityFramework para acceder a la base de datos. En este caso **NEstadoEF** actúa como un Proxy de **DEstadoEF**.

```

public class NEstadoEF : IRepository<Estados>
{
    private readonly DEstadoEF _dEstadoEF;
    0 referencias
    public NEstadoEF(DEstadoEF dEstadoEF)
    {
        _dEstadoEF = dEstadoEF;
    }
    1 referencia
    public async Task<List<Estados>> Consultar()
    {
        return await _dEstadoEF.Consultar();
    }
    1 referencia
    public async Task<Estados> Consultar(int? id)
    {
        return await _dEstadoEF.Consultar(id);
    }
    1 referencia
    public async Task<Estados> Agregar(Estados entidad)
    {
        return await _dEstadoEF.Agregar(entidad);
    }
    1 referencia
    public async Task Actualizar(Estados entidad)
    {
        await _dEstadoEF.Actualizar(entidad);
    }
    1 referencia
    public async Task Eliminar(int id)
    {
        await _dEstadoEF.Eliminar(id);
    }
}
  
```

### 4.3. Crear la Clase que consume la Web API

Crear la clase **NEstadoAPI** que implemente la Interface **IRepositorio**, y en sus métodos utilizará una Web API que expone los métodos CRUD para la clase de Entidad **Estado**

En esta clase se creará el constructor que reciba por medio de inyección de dependencias, el objeto **IConfiguration**, para leer el archivo de configuración y el objeto **HttpClient**, para consumir la Web API.

```

1 referencia
public class NEstadoAPI : IRepository<Estados>
{
    private readonly HttpClient client;
    private string _urlWebAPI;
    private IConfigurationRoot _configuration;
    //Recibe por inyección de dependencias el servicio de configuración y el httpClient
    0 referencias
    public NEstadoAPI(IConfiguration configuration, HttpClient httpClient)
    {
        _configuration = (IConfigurationRoot)configuration;
        _urlWebAPI = configuration.GetSection("urlWebAPI").Value;
        client = httpClient;
    }
    1 referencia
    public async Task<List<Estados>> Consultar()...
    1 referencia
    public async Task<Estados> Consultar(int? id)...
    1 referencia
    public async Task<Estados> Agregar(Estados estado)...
    1 referencia
    public async Task Actualizar(Estados estado)...
    1 referencia
    public async Task Eliminar(int id)...
}

public async Task<List<Estados>> Consultar()
{
    var estados = new List<Estados>();
    try
    {
        using (client)
        {
            {
                var responseTask = await client.GetAsync(_urlWebAPI);
                if (responseTask.IsSuccessStatusCode)
                {
                    var respuestaJson = await responseTask.Content.ReadAsStringAsync();
                    estados = JsonConvert.DeserializeObject<List<Estados>>(respuestaJson);
                }
                else //web api envió error de respuesta
                {
                    throw new Exception($"WebAPI. Respondio con error.{responseTask.StatusCode}");
                }
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception($"WebAPI. Respondio con error.{ex.Message}");
    }
    return estados;
}

```



```
public async Task<Estados> Consultar(int? id)
{
    var estado = new Estados();
    try
    {
        using (var client = new HttpClient())
        {
            var responseTask = await client.GetAsync(_urlWebAPI + $"/{id}");
            if (responseTask.IsSuccessStatusCode)
            {
                var respuestaJson = await responseTask.Content.ReadAsStringAsync();
                estado = JsonConvert.DeserializeObject<Estados>(respuestaJson);
            }
            else //web api envió error de respuesta
            {
                throw new Exception($"WebAPI. Respondio con error.{responseTask.StatusCode}");
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception($"WebAPI. Respondio con error.{ex.Message}");
    }
    return estado;
}

public async Task<Estados> Agregar(Estados estado)
{
    try
    {
        using (var client = new HttpClient())
        {
            HttpContent httpContent = new StringContent(JsonConvert.SerializeObject(estado), Encoding.UTF8);
            httpContent.Headers.ContentType = new MediaTypeHeaderValue("application/json");
            var responseTask = await client.PostAsync(_urlWebAPI, httpContent);
            if (responseTask.IsSuccessStatusCode)
            {
                var respuestaJson = await responseTask.Content.ReadAsStringAsync();
                estado = JsonConvert.DeserializeObject<Estados>(respuestaJson);
            }
            else //web api envió error de respuesta
            {
                throw new Exception($"WebAPI. Respondio con error.{responseTask.StatusCode}");
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception($"WebAPI. Respondio con error.{ex.Message}");
    }
    return estado;
}
```

```

public async Task Eliminar(int id)
{
    try
    {
        using (var client = new HttpClient())
        {
            var responseTask = await client.DeleteAsync(_urlWebAPI + $"/{id}");
            if (!responseTask.IsSuccessStatusCode)
            {
                throw new Exception($"WebAPI. Responadio con error.{responseTask.StatusCode}");
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception($"WebAPI. Responadio con error.{ex.Message}");
    }
}

public async Task Actualizar(Estados estado)
{
    try
    {
        using (var client = new HttpClient())
        {
            HttpContent httpContent = new StringContent(JsonConvert.SerializeObject(estado), Encoding.UTF8);
            httpContent.Headers.ContentType = new MediaTypeHeaderValue("application/json");
            var responseTask = await client.PutAsync(_urlWebAPI + $"/{estado.Id}", httpContent);
            if (!responseTask.IsSuccessStatusCode)
            {
                throw new Exception($"WebAPI. Responadio con error.{responseTask.StatusCode}");
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception($"WebAPI. Responadio con error.{ex.Message}");
    }
}

```

## 5. Crear Capa de Presentación

### 5.1. Agregar la URL en appsettings.json

Agregar un parámetro en el archivo appsettings.json que contenga la url de la web api. Debemos seguir dejando la cadena de conexión que agregamos anteriormente, ya que se requiere para el caso en que el objeto inyectado de la capa de negocio sea el accese la base de datos a través de la capa de datos.

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "InstitutoTich": "Server=MSI;Database=InstitutoTich;User ID=sa;password=1l2j1ndr1;"
  },
  "urlWebAPI": "http://localhost:5175/api/Estados"
}

```

## 5.2. Agregar el Controlador

En la carpeta Controllers, agregar un Controlador con Acciones de Lectura y Escritura



### 5.2.1. Definir Variables Globales

Definir la variable global para contener la clase de Negocio del tipo **IRepository** que suministrará el servicio de acceso a datos

```

public class EstadosController : Controller
{
    private readonly IRepository<Estados> _nEstados;
}

```

### 5.2.2. Crear el constructor

En el método constructor se recibirá por inyección de dependencias la clase de Negocio del tipo **IRepository** que suministrará el servicio de acceso a datos

```

//El constructor recibe la clase de negocio que proporciona
//el acceso a datos
0 referencias
public EstadosController(IRepository<Estados> nEstados)
{
    _nEstados = nEstados;
}

```



### 5.2.3. Cambiar el Métodos para que sean asíncronos

Agregar la palabra clave **async** para convertir los métodos en método asíncronos, lo que permitirá y obligará a usar la palabra clave **await**. Asimismo, se deben modificar para que regrese una tarea **Task<>**; ello, a excepción del método **Create**, el cual no necesita usar la palabra clave **await**.

Hay que recordar que cuando se aplica la palabra clave **await**, se suspende el método de llamada y se cede el control al autor de la llamada hasta que se completa la tarea esperada.

```
public async Task<ActionResult> Index()...
public async Task<ActionResult> Details(int? id)
public async Task<ActionResult> Create(Estados estados)
public async Task<ActionResult> Edit(int id)
public async Task<ActionResult> Edit(int id, Estados estados)
public async Task<ActionResult> Delete(int id)
public async Task<ActionResult> Delete(int id, Estados estados)
```

### 5.2.1. Crear el método privado Consultar

En virtud de que en los métodos GET:

Details(int? id)

Edit(int? id)

Delete(int? id)

se requiere presentar la información de un determinado objeto(Estado), crear un método que presente los datos el cual será invocado desde los métodos indicados anteriormente. Mismo que obtendrá el objeto a mostrar por medio del método Consultar(int id) del objeto de negocio. Con se estará reutilizando el código.

```
private async Task<ActionResult> Consultar(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var estados = await _nEstados.Consultar(id);
    if (estados == null)
    {
        return NotFound();
    }
    return View(estados);
}
```

### 5.2.2. Modificar el método Index

Modificar el método Index para que la lista de objetos a mostrar se obtenga por medio del método Consultar() del objeto de negocio.

```
public async Task<IActionResult> Index()
{
    var estados = await _nEstados.Consultar();
    return View(estados);
}
```

### 5.2.3. Modificar el método Details

Modificar el método Details para que se muestre el objeto a través del método privado Consultar(int? id)

```
public async Task<ActionResult> Details(int? id)
{
    return await Consultar(id);
}
```

### 5.2.4. Modificar el método Delete

Modificar el método Details para la solicitud GET, para que se muestre el objeto a través del método privado Consultar(int? id)

```
public async Task<ActionResult> Delete(int id)
{
    return await Consultar(id);
}
```

Modificar el método Delete para la solicitud POST para que se utilice el método Eliminar(int id) del objeto de negocio.

```
public async Task<ActionResult> Delete(int id, Estados estados)
{
    try
    {
        await _nEstados.Eliminar(id);
        return RedirectToAction(nameof(Index));
    }
    catch
    {
        return View(estados);
    }
}
```

### 5.2.5. Modificar el método Create

Modificar el método Create para la solicitud POST, a fin de que se utilice el método Agregar(Objeto objeto) del objeto de negocio.

```
public async Task<ActionResult> Create(Estados estados)
{
    try
    {
        if (ModelState.IsValid)
        {
            var edo = await _nEstados.Agregar(estados);
            return RedirectToAction(nameof(Index));
        }
        return RedirectToAction(nameof(Index));
    }
    catch
    {
        return View(estados);
    }
}
```

### 5.2.6.Modificar el método Edit

Modificar el método Edit para la solicitud GET para que el objeto a mostrar se obtenga por medio del método Consultar(int id) del objeto de negocio.

```
public async Task<ActionResult> Edit(int id)
{
    return await Consultar(id);
}
```

Modificar el método Edit para la solicitud POST para que se utilice el método Actualizar(Objeto objeto) del objeto de negocio.

```
public async Task<ActionResult> Edit(int id, Estados estados)
{
    if (id != estados.Id)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            await _nEstados.Actualizar(estados);
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!(await _nEstados.Consultar(estados.Id) != null ? true : false))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(estados);
}
```

### 5.3. Registrar los Servicios para la Inyección de Dependencias

La inyección de dependencias (en inglés Dependency Injection, DI) es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos, esto, con el objeto de disminuir el acoplamiento entre los componentes de una aplicación. Esto hace que los componentes que forman parte de la aplicación sean más independientes y, por tanto, más sencillos de mantener

ASP.NET Core admite el patrón de diseño de software de inserción de dependencias (DI), que es una técnica para conseguir la inversión de control (IoC) entre clases y sus dependencias. La inserción de dependencias en .NET es una parte integrada del framework, junto con la configuración, el registro y el patrón de opciones.

Una dependencia es un objeto del que depende otro objeto.

ASP .NET Core tiene integrado un contenedor de dependencias, también llamado inyector de dependencias, contenedor de servicios, etc.

Básicamente el contenedor de dependencias es como una caja a la que se le solicita construir los objetos. Esa caja sabe qué tiene que hacer para construir cada uno de los objetos de la aplicación. Si queremos un programador, simplemente le pedimos al contenedor de dependencias que nos cree un objeto de esa clase y él se dedica a crearlo. Finalmente, el "contenedor de dependencias" lo devuelve listo para ser usado.

En el controlador se le inyectará un objeto que implemente la interfaz `IRepositorio<Estados>`. El objetivo principal de esto es que podamos suministrar un objeto que implemente esta interfaz pero que use la clase de la capa de Datos, o bien si así nos conviene, se puede suministrar un objeto que implemente esta misma interfaz pero que el acceso a la Base de datos se hace mediante una Web API. Esto sin que tenga mayor modificación la aplicación.

También se inyectará un objeto `HttpClient`, para el caso en que la clase `IRepositorio<Estados>`. Suministrada sea la que tenga el acceso a la web api

Para registrar los servicios se hará mediante el builder, y se pueden registrar con una de las duraciones siguientes:

- Transient
- Scoped
- Singleton

Agregaremos en el archivo `program.cs` las siguientes líneas para registrar el objeto `NEstadoEF`, y `HttpClient`, Asimismo es necesario registrar el objeto `DEstadoEF`, que requiere el objeto `NEstadoEF`.

```
builder.Services.AddDbContext<EdoContext>(opcion => {
    opcion.UseSqlServer(builder.Configuration.GetConnectionString("InstitutoTich"));
});

builder.Services.AddScoped<IRepositorio<Estados>, NEstadoEF>();
builder.Services.AddHttpClient();
builder.Services.AddScoped<DEstadoEF>();

var app = builder.Build();
```

## 5.4. Agregar las Vistas

Agregar las Vistas del Controlador.

### 5.4.1.Vista Index

Agregar la Vista Index con la Plantilla List y la clase de Modelo Estados del namespace Entidades.

Agregar Vista de Razor

Nombre de vista	<input type="text" value="Index"/>
Plantilla	<input type="text" value="List"/>
Clase de modelo	<input type="text" value="Estados (Entidades)"/>
Clase DbContext	<input type="text" value=" &lt;Requerido &gt;"/>

Opciones

☐ Crear como vista parcial  
☒ Hacer referencia a bibliotecas de scripts  
☒ Usar página de diseño

...

(Dejar en blanco si se define en un archivo \_viewstart de Razor)

### 5.4.2.Vista Details

Agregar la Vista Details con la Plantilla Details y la clase de Modelo Estados del namespace Entidades.

## Agregar Vista de Razor

Nombre de vista	<input type="text" value="Details"/>
Plantilla	<input type="text" value="Details"/>
Clase de modelo	<input type="text" value="Estados (Entidades)"/>
Clase DbContext	<input type="text" value=" &lt;Requerido&gt;"/>

Opciones

☐ Crear como vista parcial  
☒ Hacer referencia a bibliotecas de scripts  
☒ Usar página de diseño

(Dejar en blanco si se define en un archivo \_viewstart de Razor)

Ajustar los AccionLink con la finalidad de concatenar al url, el id del Objeto requerido

```

@Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
@Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */ }) |
@Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })
  
```

Dejándolos de la siguiente manera

```

@Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
@Html.ActionLink("Details", "Details", new { id = item.Id }) |
@Html.ActionLink("Delete", "Delete", new { id = item.Id })
  
```

### 5.4.3.Vista Create

Agregar la Vista Create con la Plantilla Create y la clase de Modelo Estados del namespace Entidades.

## Agregar Vista de Razor

Nombre de vista

Plantilla

Clase de modelo

Clase DbContext

Opciones

☐ Crear como vista parcial

☒ Hacer referencia a bibliotecas de scripts

☒ Usar página de diseño

...

(Dejar en blanco si se define en un archivo \_viewstart de Razor)

### 5.4.4.Vista Edit

Agregar la Vista Edit con la Plantilla Edit y la clase de Modelo Estados del namespace Entidades.

## Agregar Vista de Razor

Nombre de vista

Plantilla

Clase de modelo

Clase DbContext

Opciones

☐ Crear como vista parcial

☒ Hacer referencia a bibliotecas de scripts

☒ Usar página de diseño

...

(Dejar en blanco si se define en un archivo \_viewstart de Razor)

Como normalmente no es necesario que el usuario le cambie el ID al objeto modificar el razor para el ID

```
<div class="form-group">
  <label asp-for="Id" class="control-label"></label>
  <input asp-for="Id" class="form-control" />
  <span asp-validation-for="Id" class="text-danger"></span>
</div>
```

Dejándolo de la siguiente manera



```
<div class="form-group">
  <input type="hidden" asp-for="Id" />
</div>
```

### 5.4.5.Vista Delete

Agregar la Vista Delete con la Plantilla Delete y la clase de Modelo Estados del namespace Entidades.

#### Agregar Vista de Razor

Nombre de vista	<input type="text" value="Delete"/>
Plantilla	<input type="text" value="Delete"/>
Clase de modelo	<input type="text" value="Estados (Entidades)"/>
Clase DbContext	<input type="text" value=" &lt;Requerido &gt;"/>

Opciones

☐ Crear como vista parcial  
☒ Hacer referencia a bibliotecas de scripts  
☒ Usar página de diseño

(Dejar en blanco si se define en un archivo \_viewstart de Razor)

## 5.5. Ejecutar la Aplicación Web MVC

### 5.5.1.Establecer Proyecto de Inicio

Si es necesario, establecer como Proyecto de Inicio, el Proyecto de Presentacion.

### 5.5.2.Cambiar el Enrutamiento por Default

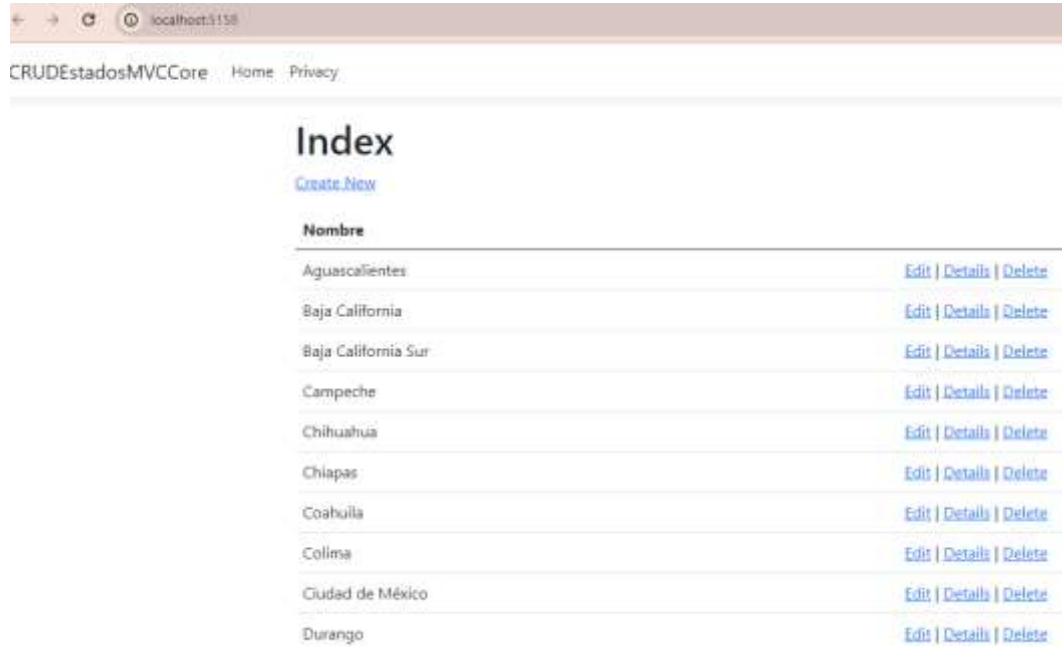
En el archivo program.cs, cambiar el enrutamiento por default, para que se dirija al Controlador Estados y la Acción Index

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Estados}/{action=Index}/{id?}");
```



### 5.5.3. Ejecutar la Aplicación

Con lo hecho anteriormente esta lista la Aplicación Web con acciones y vistas de CRUD para sea ejecutada



Nombre	
Aguascalientes	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Baja California	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Baja California Sur	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Campeche	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Chihuahua	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Chiapas	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Coahuila	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Colima	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Ciudad de México	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Durango	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

### 5.5.4. Cambiar la clase de Negocio

Ahora ejecutar la aplicación utilizando la clase de negocio **NEstadoAPI** en lugar de **NEstadoEF**, inyectando la primera en lugar de la segunda. Esto es posible ya que ambas clases implementan la misma interfaz.

En el archivo program.cs hacer la siguiente modificación:

```
builder.Services.AddDbContext<EdoContext>(opcion => {
    opcion.UseSqlServer(builder.Configuration.GetConnectionString("InstitutoTich"));
});

builder.Services.AddScoped<IRepositorio<Estados>, NEstadoEF>();
builder.Services.AddHttpClient();
builder.Services.AddScoped<DEstadoEF>();

var app = builder.Build();
```

Por lo siguiente

```
builder.Services.AddDbContext<EdoContext>(opcion => {  
    opcion.UseSqlServer(builder.Configuration.GetConnectionString("InstitutoTich"));  
});  
  
builder.Services.AddScoped<IRepositorio<Estados>, NEstadoAPI>();  
builder.Services.AddHttpClient();  
builder.Services.AddScoped<DEstadoEF>();  
  
var app = builder.Build();
```

### 5.5.5. Ejecutar la Aplicación consumiendo la API

Como la clase `NEstadoAPI` consume la Web API que cuenta con los métodos CRUD, hay que asegurarse que esta Web API se esté ejecutando. Y posteriormente ejecutar la aplicación