

CONFIGURACION Y EJEMPLO LADO BACKEND (USO DE BASE DE DATOS)

1. Creamos una clase (entidad) con el tipo de dato y las validaciones.

```
using System.ComponentModel.DataAnnotations;

namespace backEnd.Entidades
{
    5 referencias
    public class Estudiante
    {
        4 referencias
        public int Id { get; set; }

        [Required]
        [StringLength(maximumLength:20)]
        0 referencias
        public string Nombre { get; set; }
        0 referencias
        public DateTime FechaNacimiento { get; set; }
        0 referencias
        public string Sexo { get; set; }
        0 referencias
        public string Cedula { get; set; }
        0 referencias
        public string Carrera { get; set; }
    }
}
```

2. Creamos una clase llamada ApplicationDbContext y la implementamos así:

```
using backEnd.Entidades;
using Microsoft.EntityFrameworkCore;

namespace backEnd
{
    2 referencias
    public class ApplicationDbContext : DbContext
    {
        0 referencias
        public ApplicationDbContext(DbContextOptions options) : base(options)
        {
        }
        0 referencias
        public DbSet<Estudiante> Estudiantes { get; set; }
    }
}
```

NOTA: Tener instalado Microsoft.EntityFrameworkCore y Microsoft.EntityFrameworkCore.SqlServer

3. En el archivo program.cs o startup.cs agregamos el servicio :

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

NOTA: Tener instalado Microsoft.EntityFrameworkCore.Tools

4. En el archivo appsettings.json agregar las cadenas de conexión:

```
"connectionStrings": {
  "defaultConnection": "Data Source=(localdb)\\MSSQLLocalDB; Initial Catalog=UniversidadAPI; Integrated Security=True",
  "Data Source=184.168.194.75;Initial Catalog=ERPResenal;Persist Security Info=True;User ID=erpRes;Password=Host520";
}
```

NOTA: En Initial Catalog -> colocar el nombre de la bd que desea crear

5. Vamos al package manager console y digitar:

Add-Migration initial -> Crea un clase con toda la configuración de la tabla que se va a crear. Después de creada la primera tabla podemos colocar en Colegios u otra entidad en vez de initial.

Update-Database -> Crea la base de datos en Sql con la tabla Estudiantes.

NOTA: Tener instalado Microsoft.EntityFrameworkCore.Tools

6. Vamos a la clase controller, aplicamos el ApplicationDbContext, el ILogger y el IMapper y los asignamos como campo. Además, adicionamos ControllerBase y [ApiController] todo esto para que nos funcione otros métodos que implementaremos más adelante.

using Microsoft.AspNetCore.Mvc; para ControllerBase

```
namespace backEnd.Controllers
{
    [Route("api/estudiantes")]
    [ApiController]
    public class EstudiantesController : ControllerBase
    {
        private readonly ApplicationDbContext context;
        private readonly ILogger<EstudiantesController> logger;
        private readonly IMapper mapper;

        public EstudiantesController(ApplicationDbContext context, ILogger<EstudiantesController> logger,
            IMapper mapper)
        {
            this.context = context;
            this.logger = logger;
            this.mapper = mapper;
        }
    }
}
```

Nota: la variable IMapper no la creamos todavía ya que tenemos que seguir primero los pasos del 7 al 10.

7. Vamos a implementar DTO(Data Transfer Object) para no exponer la entidad en el controller. Creamos un archivo EstudianteDTO y EstudianteCreacionDTO con la misma configuración de la entidad Estudiante.

```
using System.ComponentModel.DataAnnotations;

namespace backEnd.DTOs
{
    public class EstudianteCreacionDTO
    {
        [Required]
        [StringLength(maximumLength: 20)]
        public string Nombre { get; set; }
        public DateTime FechaNacimiento { get; set; }
        public string Sexo { get; set; }
        public string Cedula { get; set; }
        public string Carrera { get; set; }
    }
}

namespace backEnd.DTOs
{
    public class EstudianteDTO
    {
        public int Id { get; set; }
        public string Nombre { get; set; }
        public DateTime FechaNacimiento { get; set; }
        public string Sexo { get; set; }
        public string Cedula { get; set; }
        public string Carrera { get; set; }
    }
}
```

8. Instalar AutoMapper para mapear estudiantes.



9. En el archivo program.cs o startup.cs agregamos el servicio:

```
builder.Services.AddAutoMapper(typeof(Program));
```

10. Creamos el archivo AutoMapperProfiles.cs y lo heredamos de " :Profile " y agregamos el mapeo de estudiante y lo hacemos de doble vía para consulta, también hacemos uno para creación.

```

using AutoMapper;
using backEnd.DTOS;
using backEnd.Entidades;

namespace backEnd.Utilidades
{
    1 referencia
    public class AutoMapperProfiles: Profile
    {
        0 referencias
        public AutoMapperProfiles()
        {
            CreateMap<Estudiante, EstudianteDTO>().ReverseMap();
            CreateMap<EstudianteCreacionDTO, Estudiante>();
        }
    }
}

```

11. Ahora implementamos los métodos get, post, put, delete para interactuar con la BD. El context es la palabra que sirve para hacer el crud en SQL. Importante usar async Task y await para que realice otras tareas la aplicación mientras se realiza el crud.

Nota: Tener [using Microsoft.EntityFrameworkCore](#); para usar los query

```

[HttpGet("{id:int}")]
0 referencias
public async Task<ActionResult<EstudianteDTO>> Get(int id)
{
    var estudiante = await context.Estudiantes.FirstOrDefaultAsync(x => x.Id == id);
    if (estudiante == null)
        return NotFound();

    return mapper.Map<EstudianteDTO>(estudiante);
}

[HttpGet]
0 referencias
public async Task<ActionResult<List<EstudianteDTO>>> Get()
{
    var estudiante = await context.Estudiantes.ToListAsync();
    return mapper.Map<List<EstudianteDTO>>(estudiante);
}

[HttpPost]
0 referencias
public async Task<ActionResult> Post([FromBody] EstudianteCreacionDTO estudianteCreacionDTO)
{
    var estudiante = mapper.Map<Estudiante>(estudianteCreacionDTO);
    context.Add(estudiante);
    await context.SaveChangesAsync();
    return NoContent();
}

[HttpPut("{id:int}")]
0 referencias
public async Task<ActionResult> Put(int Id, [FromBody] EstudianteCreacionDTO estudianteCreacionDTO)
{
    var estudiante = await context.Estudiantes.FirstOrDefaultAsync(x => x.Id == Id);
    if (estudiante == null)
        return NotFound();
    estudiante = mapper.Map(estudianteCreacionDTO, estudiante);
    await context.SaveChangesAsync();
    return NoContent();
}

[HttpDelete("{id:int}")]
0 referencias
public async Task<ActionResult> Delete(int id)
{
    var existe = await context.Estudiantes.AnyAsync(x => x.Id == id);
    if (existe == false)
        return NotFound();

    context.Remove(new Estudiante() { Id = id });
    await context.SaveChangesAsync();
    return NoContent();
}

```

12. Algunas reglas de ruteo

- [HttpGet("todos")] → api/estudiantes/
- [HttpGet("{Id:int}")] → api/estudiantes/1
- [HttpGet("{Id}/{nombre}")] → api/estudiantes/2/carlos
- [HttpGet("{Id:int}/{nombre=luis}")] → api/estudiantes/2, el campo nombre viene por defecto.
- [HttpGet("/listadogeneros")] → /listadogeneros, se quita el api/géneros

13. ActionResult // se retorna dato de la clase o un dato que herede de ActionResult (Puede ser un error Ej. NotFound()).

14. IActionResult // se retorna cualquier tipo de dato string, int, clases, etc.. y no deja pasar un parámetro de clase en su definición.

15. Programación asíncrona: en la declaración y desarrollo del método se coloca de la siguiente manera:

- Public async Task<T> nombreFuncion()
- Await → se coloca dentro del método antes de ejecutar un código que genere demora.
- Si llamamos este método desde otro método este también debe ser async Task<> y dentro debe tener también el await al llamar el método.

16. Query Strings

• api/generos?id=5&apellido=Gavilán

```
[HttpGet]
public ActionResult<Genero> Get(int Id, string apellido)
{...}
```

17. Valores del Formulario

```
[HttpPost]
public void Post([FromBody] Genero genero)
{...}
```

18. El [BindRequired] obliga a la petición http enviar el nombre, y en el return envía el error.

```
[HttpGet("{Id:int}")] // api/generos/3/felipe
public async Task<ActionResult<Genero>> Get(int Id, [BindRequired] string nombre)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
}
```

19. [ApiController] : Si la petición http tiene errores nos retorna el error sin necesidad de poner validaciones en el método, y para que esto funcione en la definición de la entidad tenemos que crear validaciones y

podemos también personalizar los mensajes del error.

```
public class Genero
{
    public int Id { get; set; }
    [Required(ErrorMessage = "El campo {0} es requerido")]
    public string Nombre { get; set; }
}

[HttpPost]
public ActionResult Post([FromBody] Genero genero)
{
    return NoContent();
}
```