Desarrollo de Software Empresarial

CICLO 02 -2024

Guía 9: Pruebas Unitarias

Competencias

- Que el estudiante aprenda a como0 agregar pruebas unitarias a un proyecto en :NET.
- Que el estudiante aprenda a montar una aplicacion en un contenedor.

Introducción Teórica

¿Qué son las pruebas unitarias?

Las pruebas unitarias son un tipo de prueba de software que se centra en verificar el funcionamiento correcto de una unidad individual de código, como una función, un método o una clase. La idea es comprobar que cada unidad de código realiza las tareas para las que fue diseñada y produce los resultados esperados bajo diversas condiciones.

Ventajas de las pruebas unitarias

- Detección Temprana de Errores: Permiten identificar errores en las primeras etapas del desarrollo, lo que facilita la corrección y evita problemas en fases posteriores.

- Facilitan el Refactorizado: Al tener un conjunto de pruebas unitarias, puedes refactorizar el código con mayor confianza, sabiendo que las pruebas te ayudarán a asegurar que los cambios no introduzcan errores inesperados.

- Documentación del Código: Las pruebas unitarias sirven como una forma de documentación del comportamiento esperado del código. Pueden ayudar a otros desarrolladores a entender cómo debería funcionar una unidad específica del sistema.

- Facilitan el Desarrollo Ágil: Son fundamentales en prácticas de desarrollo ágil como TDD (Test-Driven Development), donde se escriben las pruebas antes de implementar la funcionalidad.

- Mejoran la Calidad del Código: Al enfocarse en unidades individuales, las pruebas unitarias fomentan la escritura de código más modular y menos dependiente, lo que puede llevar a un diseño más limpio y mantenible.

- Reducción de Costos: Aunque escribir pruebas unitarias puede requerir tiempo inicial, a largo plazo pueden reducir el costo total de desarrollo al minimizar el tiempo necesario para encontrar y corregir errores.

¿Por qué incluirlas en el desarrollo de software?

- Prevención de Regresiones: Las pruebas unitarias aseguran que los cambios en el código no rompan funcionalidades existentes. Esto es crucial para mantener la estabilidad del software a medida que evoluciona.

- Mejoras Continuas: Con pruebas unitarias, puedes mejorar continuamente el código sin miedo a introducir nuevos errores. Esto es especialmente útil en proyectos de largo plazo.

- Mayor Confianza en el Código: Proporcionan una base sólida para verificar que el código funciona como se espera, lo que aumenta la confianza en la calidad del software entregado.

- Automatización de Pruebas: Las pruebas unitarias son fáciles de automatizar, lo que permite realizar pruebas frecuentes y consistentes sin intervención manual. Esto es especialmente útil en integración continua (CI) y despliegues continuos (CD).

- Facilitan el Desarrollo Colaborativo: En equipos grandes, las pruebas unitarias ayudan a garantizar que las diferentes partes del software funcionen juntas como se espera, lo que facilita el trabajo en equipo y la integración de componentes.

Test-Driven Development (TDD)

El Test-Driven Development (TDD) es una metodología de desarrollo de software que se centra en escribir pruebas antes de escribir el código funcional. Es una técnica que ayuda a garantizar que el software cumple con los requisitos desde el principio y facilita la detección de errores de manera temprana en el ciclo de desarrollo.

El proceso de TDD generalmente sigue estos pasos:

- Escribir una prueba: Antes de escribir el código de la funcionalidad, se crea una prueba automatizada que define cómo debe comportarse la función o el componente que se va a desarrollar. Esta prueba inicialmente fallará porque la funcionalidad aún no está implementada.

- Escribir el código mínimo necesario para pasar la prueba: Se implementa el código más simple y mínimo que permita que la prueba pase. El enfoque está en escribir solo lo necesario para que la prueba tenga éxito.

- Refactorizar: Una vez que la prueba pasa, se mejora o refactoriza el código para hacerlo más limpio y eficiente, manteniendo todas las pruebas funcionando correctamente.

- Repetir: Se repite el ciclo creando nuevas pruebas para nuevas funcionalidades, escribiendo el código correspondiente y refactorizando según sea necesario.

Estructura de una prueba unitaria

- **Arrange:** establecer condiciones iniciales, preparando el escenario, declarar variables e instancias objetos (de ser necesario).

- **Act:** ejecución del fragmento de código de la prueba y del fragmento a testear.

- **Assert:** comprobación del resultado obtenido.

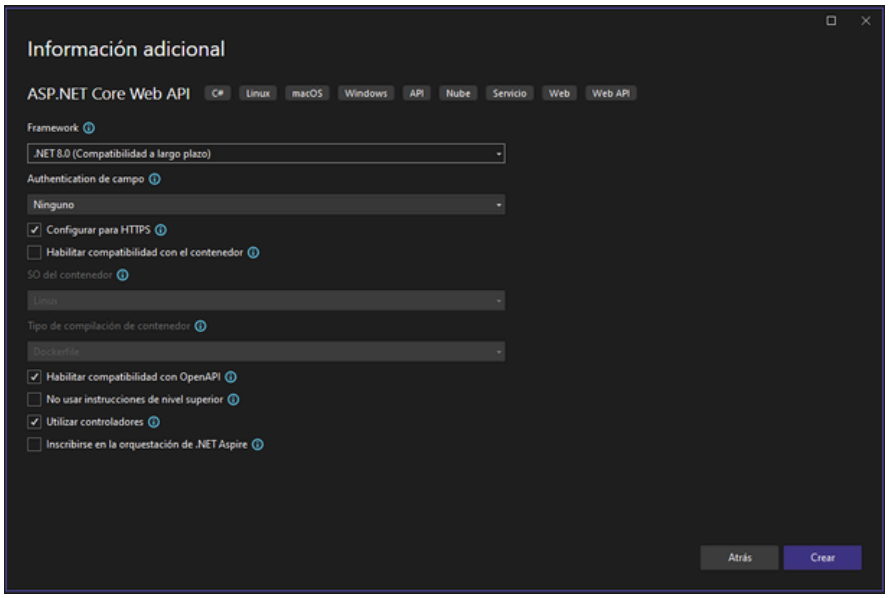
Buenas practicas en las pruebas unitarias

- No se debe de probar más de una cosa en una sola prueba, es mejor hacer un test por validación.
- Deben de ser de rápida ejecución.
- Resultados consistentes.
- Deben de ser aisladas, no depender de otra pruebas u otros agentes externos.
- El fallo de una prueba debe de ser fácilmente reconocible.
- Sencillas y ágiles.
- Veracidad en la pruebas, muchas veces se logra induciendo un fallo.

Procedimiento

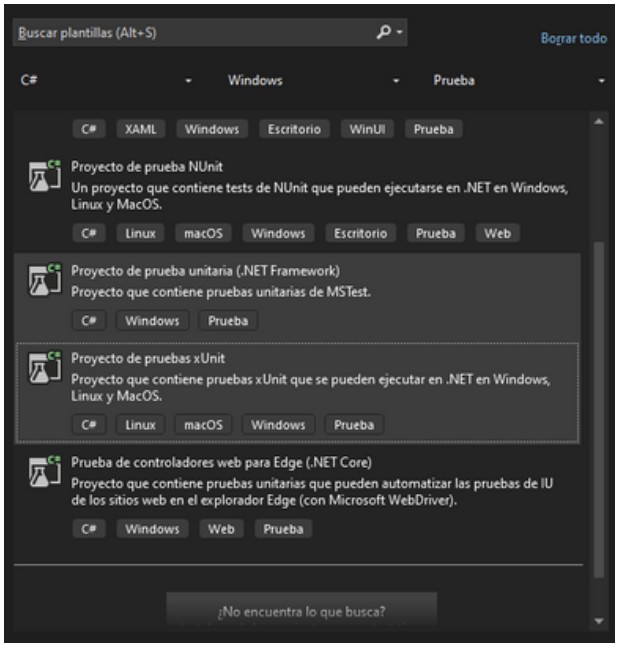
Creación del proyecto Web API

1. Crear un proyecto nuevo del tipo **Web API** con nombre **LibrosAPI**.

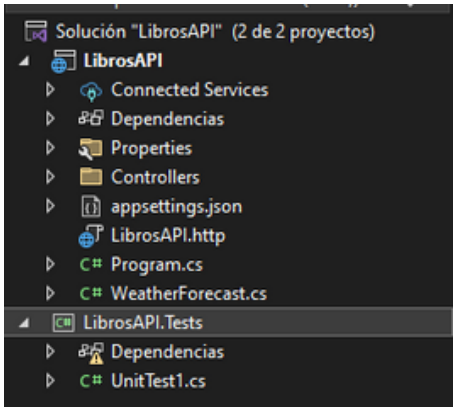


Creación del proyecto de pruebas unitarias.

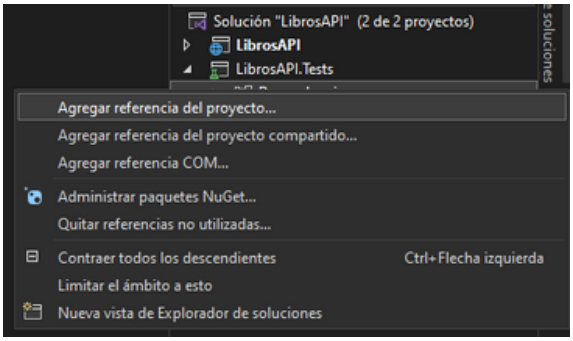
1. Necesitamos crear el proyecto de tipo **Proyecto de pruebas xUnit** con nombre **LibrosAPI.Tests** dentro de la misma solución del proyecto **LibrosAPI**. Para ello en el explorador de soluciones damos click derecho sobre la solución y elegimos la opción **Agregar > Nuevo Proyecto**.

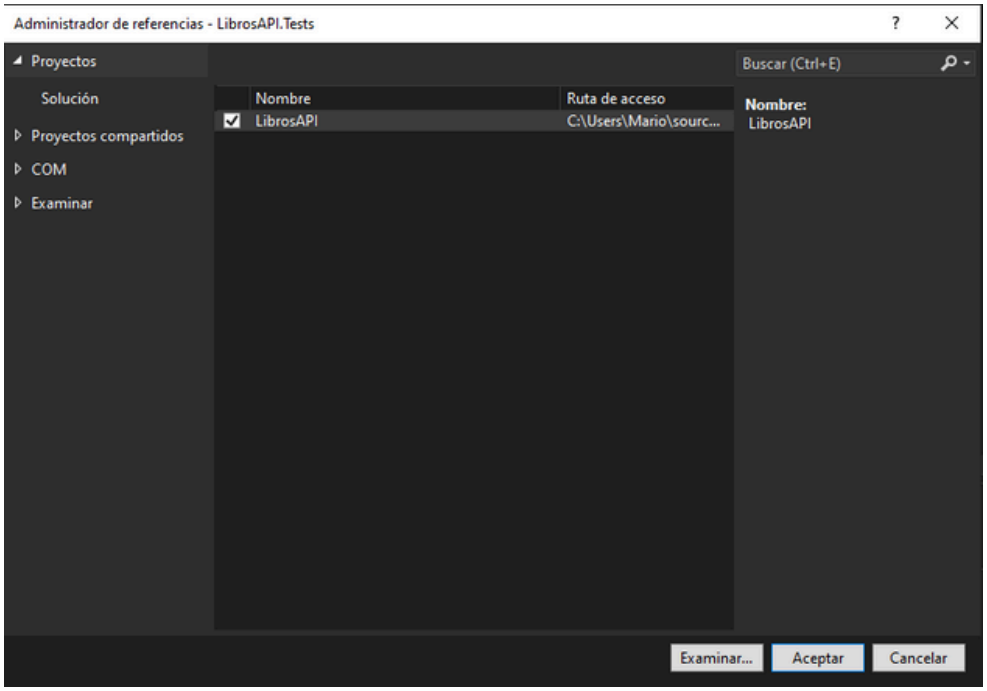


La solución debe quedar estructurada de la siguiente manera:



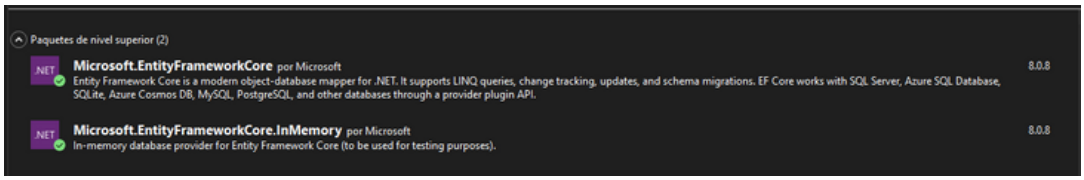
2. En el proyecto **LibrosAPI.Tests** agregamos la referencia al proyecto de pruebas **LibrosAPI**.





3. En el proyecto **LibrosAPI** agregar los paquetes:

- **Microsoft.EntityFrameworkCore**
- **Microsoft.EntityFrameworkCore.InMemory**



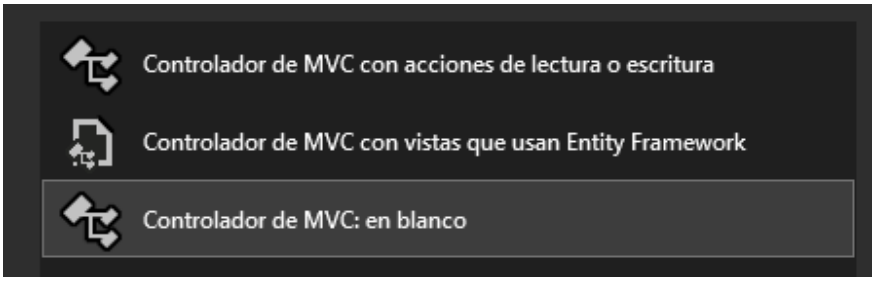
4. Agregar el modelo Libros a nuestro proyecto **LibrosAPI** en la carpeta **Models.**

```
1 public class Libro
2 {
3     public int Id { get; set; }
4     public string Titulo { get; set; }
5     public string Autor { get; set; }
6     public DateTime FechaPublicacion { get; set; }
7 }
```

5. Configuramos el **DbContext**.

```
1 using Microsoft.EntityFrameworkCore;
2
3 namespace LibrosAPI.Models
4 {
5     public class LibrosDbContext : DbContext
6     {
7         public LibrosDbContext(DbContextOptions<LibrosDbContext> options) : base(options) { }
8
9         public DbSet<Libro> Libros { get; set; }
10    }
11 }
```

6. Creamos un controlador en blanco y agregamos los siguientes métodos que nos servirán para escribir las pruebas.



```
using LibrosAPI.Models;
using Microsoft.AspNetCore.Mvc;

namespace LibrosAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class LibrosController : ControllerBase
    {
        private readonly LibrosDbContext _context;

        public LibrosController(LibrosDbContext context)
        {
            _context = context;
        }

        [HttpPost]
        public async Task<ActionResult<Libro>> PostLibro(Libro libro)
        {
            if (string.IsNullOrEmpty(libro.Titulo))
            {
                return BadRequest("El título no puede estar vacío.");
            }

            _context.Libros.Add(libro);
            await _context.SaveChangesAsync();
            return CreatedAtAction(nameof(GetLibro), new { id = libro.Id }, libro);
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<Libro>> GetLibro(int id)
        {
            var libro = await _context.Libros.FindAsync(id);
            if (libro == null)
            {
                return NotFound();
            }
            return libro;
        }
    }
}
```

Si queremos probar nuestros dos endpoints es necesario agregar el archivo **Program.cs** (ver imagen de ejemplo) la configuración para especificar el contexto de la base de datos, para este caso no lo haremos debido a que lo configuraremos en el proyecto de pruebas.

```
// Add services to the container.
builder.Services.AddDbContext<LibrosDbContext>(options => options.UseInMemoryDatabase("LibrosInMemoryDb"));
```

Implementación de Pruebas Unitarias

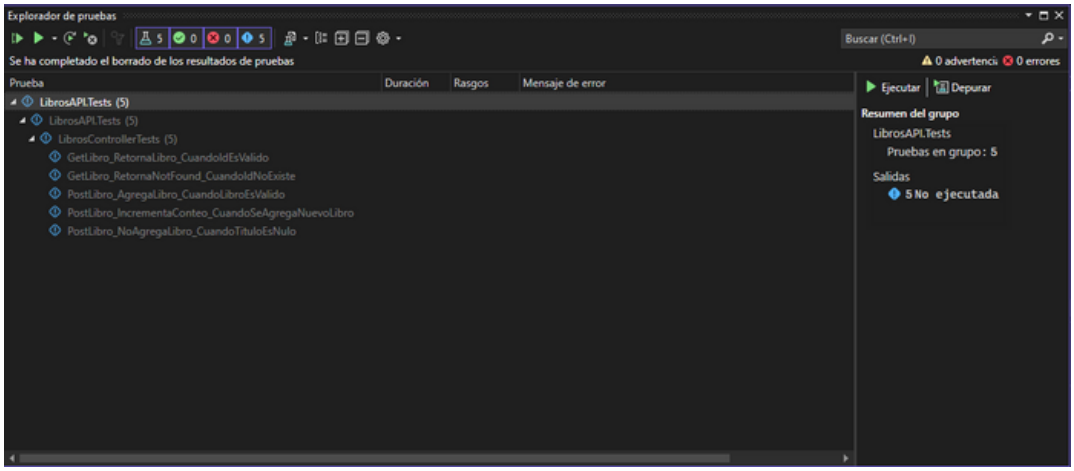
1. Agregar la clase **Setup** en el proyecto **LibrosAPI.Tests** con el siguiente código:

```
using LibrosAPI.Models;
using Microsoft.EntityFrameworkCore;

namespace LibrosAPI.Tests
{
    public static class Setup
    {
        public static LibrosDbContext GetInMemoryDatabaseContext()
        {
            var options = new DbContextOptionsBuilder<LibrosDbContext>()
                .UseInMemoryDatabase(databaseName: Guid.NewGuid().ToString())
                .Options;

            var context = new LibrosDbContext(options);
            context.Database.EnsureCreated();
            return context;
        }
    }
}
```

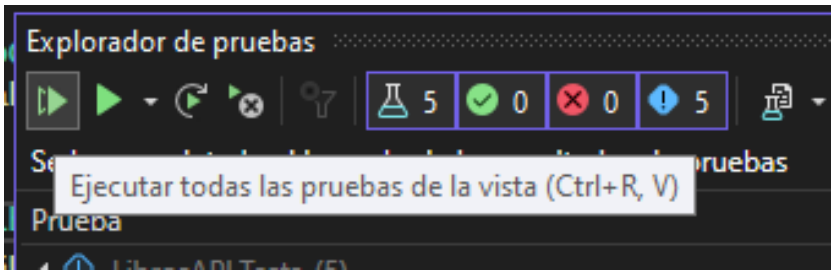
2. En el proyecto **LibrosAPI.Tests** agregar la clase **LibrosControllerTests** con el código:

```
using LibrosAPI.Controllers;
using LibrosAPI.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace LibrosAPI.Tests
{
    public class LibrosControllerTests
    {
        [Fact]
        public async Task PostLibro_AgregaLibro_CuandoLibroEsValido()
        {
            // Arrange
            var context = Setup.GetInMemoryDatabaseContext();
            var controller = new LibrosController(context);
            var nuevoLibro = new Libro { Titulo = "El Principito", Autor = "Antoine de Saint-Exupéry",
FechaPublicacion = new DateTime(1943, 4, 6) };

            // Act
            var result = await controller.PostLibro(nuevoLibro);

            // Assert
            var createdResult = Assert.IsType<CreatedAtActionResult>(result.Result);
            var libro = Assert.IsType<Libro>(createdResult.Value);
            Assert.Equal("El Principito", libro.Titulo);
        }

        [Fact]
        public async Task GetLibro_RetornaLibro_CuandoIdEsValido()
        {
            // Arrange
            var context = Setup.GetInMemoryDatabaseContext();
            var controller = new LibrosController(context);
            var libro = new Libro { Titulo = "1984", Autor = "George Orwell", FechaPublicacion = new
DateTime(1949, 6, 8) };
            context.Libros.Add(libro);
            await context.SaveChangesAsync();

            // Act
            var result = await controller.GetLibro(libro.Id);

            // Assert
            var actionResult = Assert.IsType<ActionResult<Libro>>(result);
            var returnValue = Assert.IsType<Libro>(actionResult.Value);
            Assert.Equal("1984", returnValue.Titulo);
        }

        [Fact]
        public async Task GetLibro_RetornaNotFound_CuandoIdNoExiste()
        {
            // Arrange
            var context = Setup.GetInMemoryDatabaseContext();
            var controller = new LibrosController(context);

            // Act
            var result = await controller.GetLibro(999);

            // Assert
            Assert.IsType<NotFoundResult>(result.Result);
        }

        [Fact]
        public async Task PostLibro_NoAgregaLibro_CuandoTituloEsNulo()
        {
            // Arrange
            var context = Setup.GetInMemoryDatabaseContext();
            var controller = new LibrosController(context);
            var nuevoLibro = new Libro { Titulo = null, Autor = "Autor", FechaPublicacion = new DateTime(2000, 1,
1) };

            // Act
            var result = await controller.PostLibro(nuevoLibro);

            // Assert
            Assert.IsType<BadRequestObjectResult>(result.Result);
        }

        [Fact]
        public async Task PostLibro_IncrementaConteo_CuandoSeAgregaNuevoLibro()
        {
            // Arrange
            var context = Setup.GetInMemoryDatabaseContext();
            var controller = new LibrosController(context);
            var libroInicial = new Libro { Titulo = "Libro 1", Autor = "Autor 1", FechaPublicacion = DateTime.Now
};
            await controller.PostLibro(libroInicial);

            var nuevoLibro = new Libro { Titulo = "Libro 2", Autor = "Autor 2", FechaPublicacion = DateTime.Now };

            // Act
            await controller.PostLibro(nuevoLibro);
            var libros = await context.Libros.ToListAsync();

            // Assert
            Assert.Equal(2, libros.Count);
        }
    }
}
```

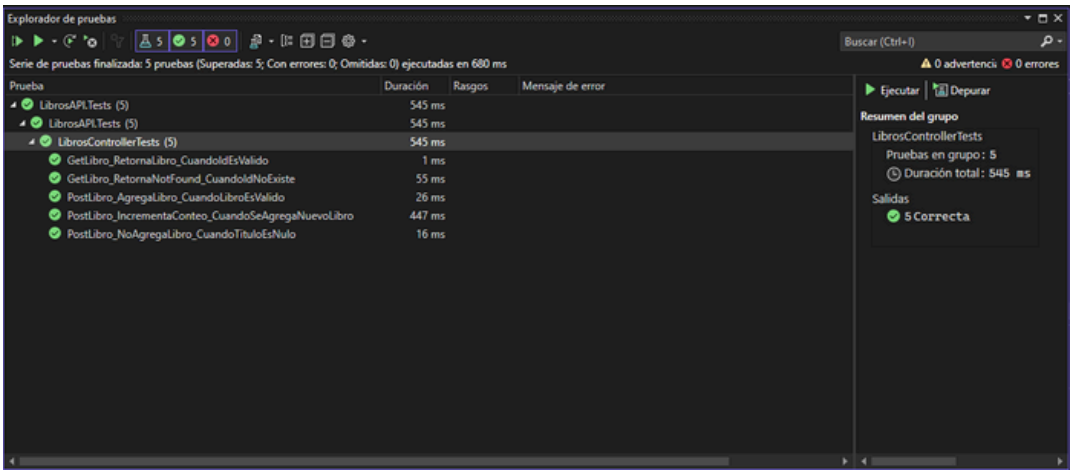
3. Para ejecutar las pruebas en Visual Studio debemos ir a la opción **Pruebas > Explorador de Pruebas**. Se mostrara una ventana con todos los archivos en los que se encuentren pruebas definidas, para nuestro caso solo aparecerá el archivo **LibrosControllerTest**.



4. En esta ventana se pueden ejecutar todas las pruebas o solo las que se selecciones, para ejecutarlas todas damos click en el botón **Ejecutar todas las pruebas de la vista.**



5. En la ventana podremos visualizar el resumen de la ejecución de las pruebas, cuales son las que fueron exitosas y cuales no.



Las pruebas escritas se definen de la siguiente manera:

- **PostLibro_AgregaLibro_CuandoLibroEsValido**: Verifica que un libro válido se agregue correctamente.

- **GetLibro_RetornaLibro_CuandoIdEsValido**: Comprueba que se retorne el libro correcto cuando se busca por un ID válido.

- **GetLibro_RetornaNotFound_CuandoIdNoExiste**: Verifica que se retorne un NotFound cuando se busca un libro con un ID inexistente.

- **PostLibro_NoAgregaLibro_CuandoTituloEsNulo**: Asegura que no se agregue un libro si el título es nulo (requiere modificación de validaciones en el controlador para este caso).

- **PostLibro_IncrementaConteo_CuandoSeAgregaNuevoLibro**: Comprueba que el conteo de libros en la base de datos aumente al agregar un nuevo libro.

Desarrollo de Habilidades

Ejercicio 1

Desarrolla una aplicación en ASP .NETCore para gestionar el registro de personas naturales.

Campos mínimos requeridos

Personas:

- Id
- Primer nombre
- Segundo nombre
- Primer apellido
- Segundo nombre
- DUI
- Fecha de nacimiento

Además, considere las siguiente validaciones:

- Primero nombre y primer apellido son requeridos.
- Segundo nombre y segundo apellido son opciones.
- Tanto los nombres como los apellidos no pueden exceder las longitud de 100 caracteres.
- Fecha de nacimiento no puede ser nula y debe ser una fecha valida.
- Validar el formato del DUI (01234567-8)

Escriba las pruebas correspondientes para comprobar que el controlador escrito en la API trabaje de manera adecuada.