30) Prueba técnica Devontic

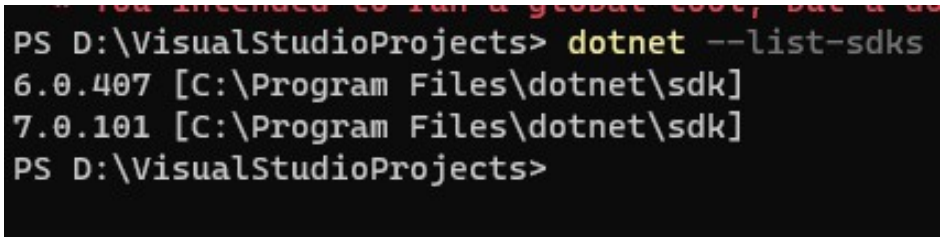
1) Revisando el SDK

Usualmente .NET está instalado en Windows para revisar las versiones actualmente instaladas, ejecutamos el siguiente comando en una ventana de PowerShell.

PowerShell

```
dotnet --list-sdks
```

Este nos mostrara las versiones actulamente instaladas de .NET SDK



Para esta prueba usaré la versión actualmente instalada, la **6.0.407.**

2) Creando el proyecto

Para crear el proyecto abriremos una consola de PowerShell y ejecutaremos los siguientes comandos.

PowerShell

```
1
2  dotnet new globaljson --sdk-version 6.0.407 --output Devontic/SimpleApp
3
4  dotnet new web --no-https --output Devontic/SimpleApp --framework net6.0
5
6  dotnet new sln -o Devontic
7
8  dotnet sln Devontic add Devontic/SimpleApp
9
```

Estos comandos agregarán crearan y agregaran nuestro proyecto a una solución.

3) Creando el proyecto Unit Test

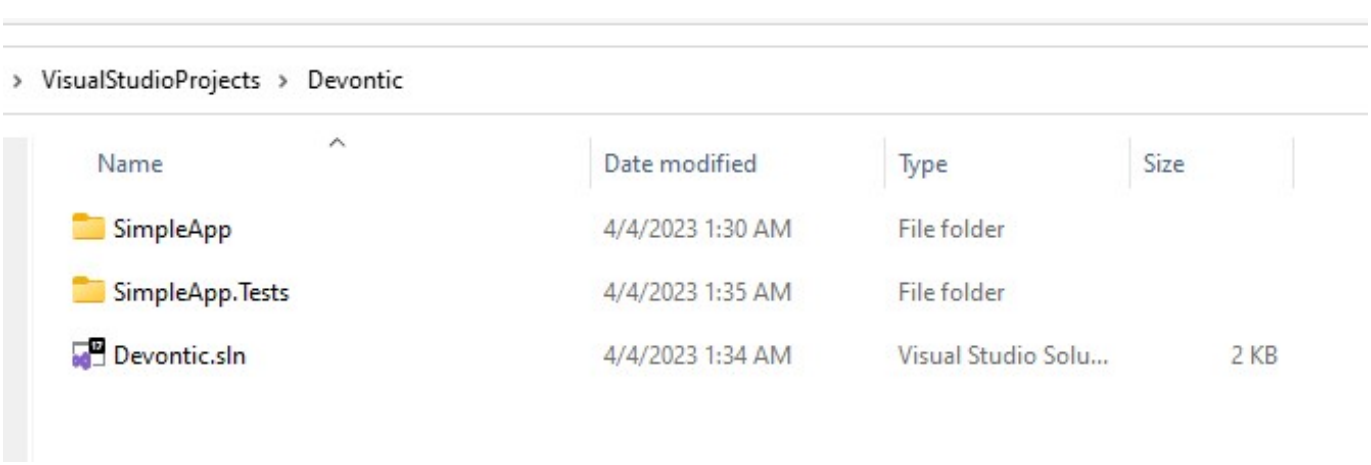
Crearemos otro proyecto para hacer pruebas unitarias.
Uno generalmente crea un proyecto de Visual Studio separado para los **Unit test**.
La convención es nombrar el proyecto de Unit Testing
como **<NombreAplicación>.tests**.

Ejecuta los siguientes comandos dentro de la carpeta testing.

PowerShell

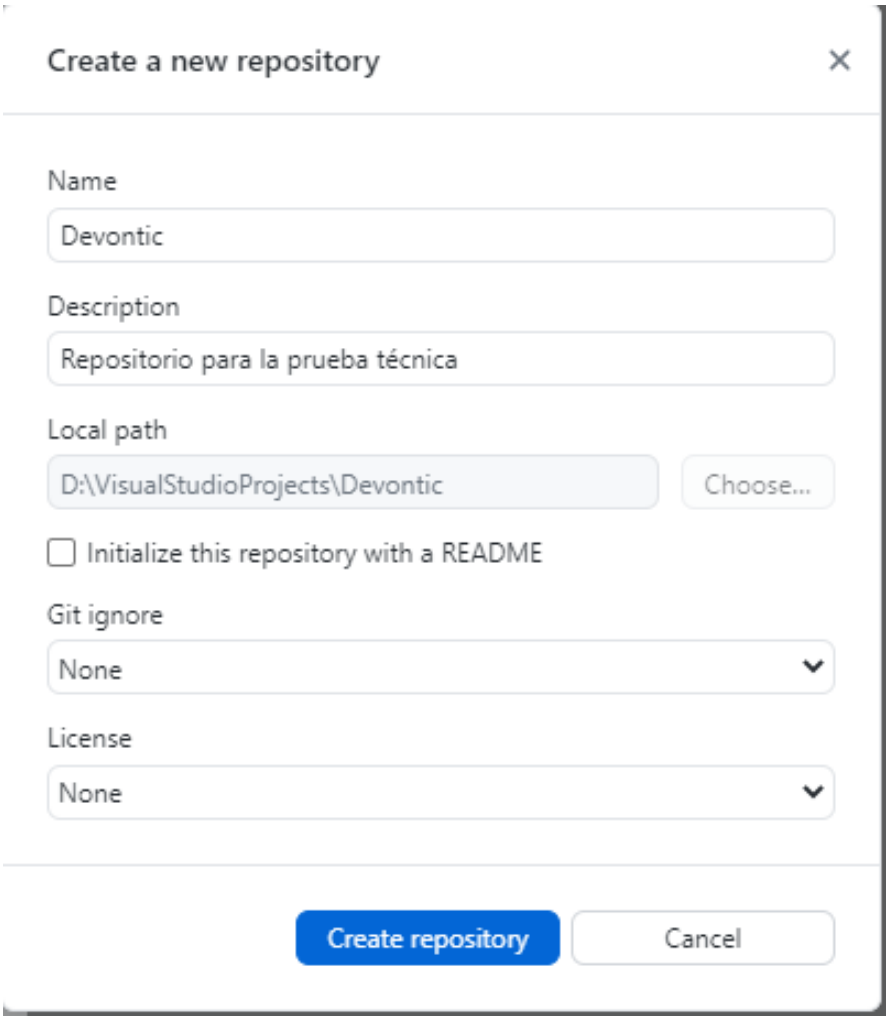
```
1  dotnet new xunit -o SimpleApp.Tests --framework net6.0
2
3  dotnet sln add SimpleApp.Tests
4
5  dotnet add SimpleApp.Tests reference SimpleApp
```

Como resultado tendremos un proyecto parecido a esto.
Este se creara con una clase **UnitTest1**, la borramos para evitar confusiones.

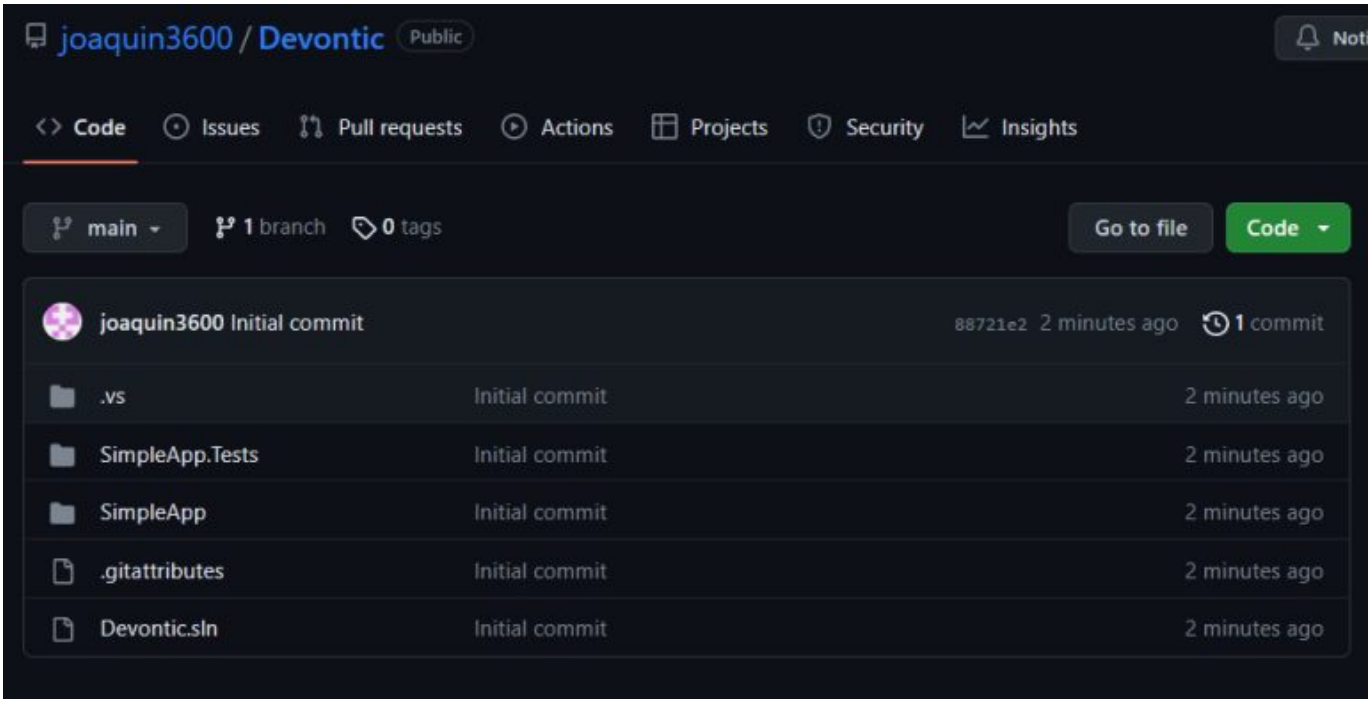


4) Creando el repositorio

Creamos el repositorio por medio de **GitHub Desktop**, seleccionado la carpeta donde está nuestra solución.

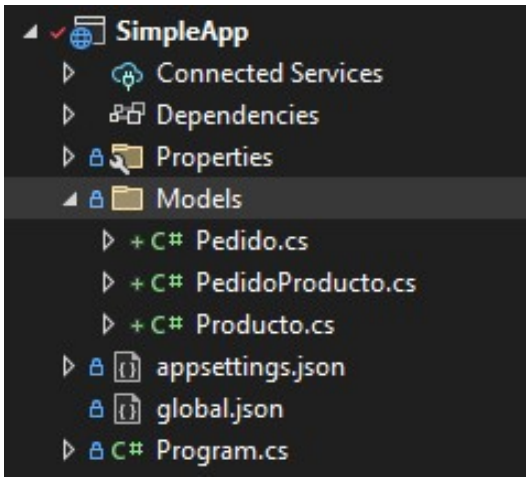


Al terminar este podemos publicar el repositorio y ya estará en nuestra Web.



5) Creando el modelo de datos

En esa sección crearemos tres clases relacionadas **Pedido**, **PedidoProducto** y **Producto**.
Primeramente, crearemos una carpeta llamada **Models** en el proyecto **SimpleApp** y crearemos las siguientes clases dentro de esta.



Pedido.cs

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace SimpleApp.Models
{
    public class Pedido
    {
        public long PedidoId { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode
        public DateTime Fecha { get; set; }
```

Producto.cs

C#

```
using System.ComponentModel.DataAnnotations.Schema;
namespace SimpleApp.Models
{
    public class Producto
    {
        public long         ProductoId   { get; set; }
        public string       Nombre       { get; set; } = string.Empty;
        [Column(TypeName = "decimal(8, 2)")]
        public decimal      Precio       { get; set; }

    }
```

PedidoProducto.cs

C#

```
using System.ComponentModel.DataAnnotations.Schema;

namespace SimpleApp.Models
{
    public class PedidoProducto
    {
        public long             PedidoProductoId    { get; set; }

        public long?            ProductoId          { get; set; }
        [ForeignKey("ProductoId")]
        public virtual Producto Producto            { get; set; }
```

Cada una de las tres clases del modelo de datos define una **propiedad clave** cuyo valor será asignado por la base de datos cuando se almacenan nuevos objetos.
La propiedad **Precio** se ha decorado con el atributo **Columna**, que especifica la precisión de la
valores que serán almacenados en la base de datos.

6) Instalando la herramienta global para manejar bases de datos.

Entity framework requiere un paquete llamado ***Global tool*** que es usado para manejar bases de datos por medio de la línea de comandos.
Este paquete provee los comandos ***dotnet ef***.
Para instalarlo corre los siguientes comandos.

PowerShell

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 6.0.0
```

El primer comando remueve cualquier versión existente del proyecto y el siguiente instala la versión requerida para este proyecto.
Para asegurarnos de que el paquete funciona como es esperado, ejecutemos el siguiente comando.

PowerShell

```
dotnet ef --help
```

7) Instalando Entity Framework Core

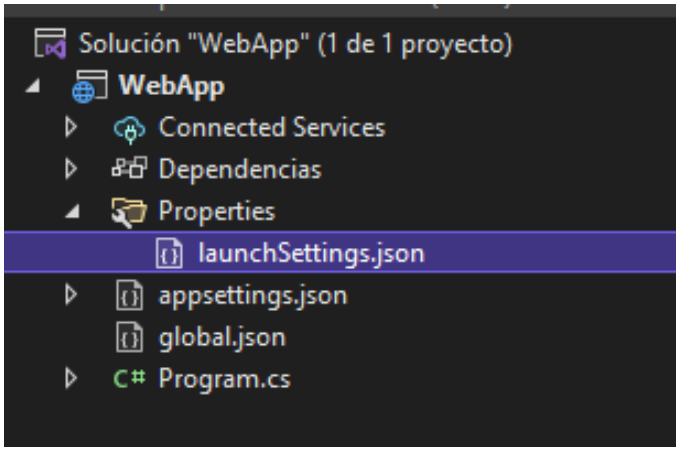
Entity framework requiere de algunos **paquetes** que sean añadidos al proyecto. Para esto ejecuta los siguientes comandos dentro de la carpeta del proyecto.

PowerShell

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 6.0.0
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 6.0.0
```

7.5) Actualizando puertos

Actualizamos los puertos del proyecto, modificando el archivo **launchSettings.json**.



JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {

",
      "sslPort": 0
    }
  },
  "profiles": {
```

8) Crear el contexto de la base de datos

Para crear la clase de contexto **Entity Framework Core** que proporcionará acceso a la base de datos, agregue un archivo llamado **DataContext.cs** a la carpeta **Models** y agregue el código que se muestra a continuación.
La clase de contexto define las propiedades que se utilizarán para consultar la base de datos sobre **Producto**, **Pedido** y **ProductoPedido**.

C#

```
using Microsoft.EntityFrameworkCore;
namespace SimpleApp.Models
{
    public class DataContext : DbContext
    {
        public DataContext(DbContextOptions<DataContext> opts) : base(opts)
        public DbSet<Pedido>           Pedidos         => Set<Pedido>();
        public DbSet<Producto>         Productos       => Set<Producto>();
        public DbSet<PedidoProducto>   PedidoProductos => Set<PedidoProduc
    }
```

9) Preparando los primeros datos

C#

```
using Microsoft.EntityFrameworkCore;
using SimpleApp.Models;
using System.ComponentModel.DataAnnotations;

namespace WebApp.Models
{
    public static class SeedData
    {
        public static void SeedDatabase(DataContext context)
        {
            context.Database.Migrate();
            if (context.Productos.Count() == 0 && context.Pedidos.Count() == 0
            {
                Producto p1 = new Producto { Nombre = "Asiento",            Pr
                Producto p2 = new Producto { Nombre = "Chaqueta",           Pr

                Producto p3 = new Producto { Nombre = "Pelota",             Pr
                Producto p4 = new Producto { Nombre = "Banderas de campo",  Pr

                Producto p5 = new Producto { Nombre = "Estadio",            Pr
                Producto p6 = new Producto { Nombre = "Gorra",              Pr

                Pedido pe1 = new Pedido { Fecha = DateTime.Parse("2023-05-02")
                Pedido pe2 = new Pedido { Fecha = DateTime.Parse("2022-05-02")
                Pedido pe3 = new Pedido { Fecha = DateTime.Parse("2021-05-02")

                context.PedidoProductos.AddRange(
                    new PedidoProducto { Pedido = pe1, Producto = p1 },
                    new PedidoProducto { Pedido = pe1, Producto = p2 },

                    new PedidoProducto { Pedido = pe2, Producto = p3 },
                    new PedidoProducto { Pedido = pe2, Producto = p4 },

                    new PedidoProducto { Pedido = pe3, Producto = p5 },
                    new PedidoProducto { Pedido = pe3, Producto = p6 }
                );
                context.SaveChanges();
            }
        }
    }
}

```

10) Configurar los servicios de Entity Framework

Modifica la clase **Program.cs** con el siguiente código para configurar los servicios que serán usados para acceder a la base de datos.

C#

```
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<DataContext>(opts =>
    {
         opts.UseSqlServer(builder.Configuration[
         "ConnectionStrings:ProductConnection"]);
         opts.EnableSensitiveDataLogging(true);
    });

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

var context = app.Services.CreateScope().ServiceProvider.GetRequiredService<Da
SeedData.SeedDatabase(context);
app.Run();
```

11) Definiendo la cadena de conexión

Para definir la cadena de conexión que se utilizará para los datos de la aplicación, agregue la configuración que se muestra a continuación en el archivo **appsettings.json**.
La cadena de conexión debe introducirse en una línea sola.
Además de la cadena de conexión, se establece los detalles de registro para **Entity Framework Core** para que se registren las consultas SQL enviadas a la base de datos.

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings":
   {
     "ProductConnection": "Server=(localdb)\\MSSQLLocalDB;Database=Pedidos;Mul
   }
 }

```

12) Creando y aplicando la migración

A continuación crearemos una migración de .NET.

PowerShell

```
dotnet ef migrations add Initial
```

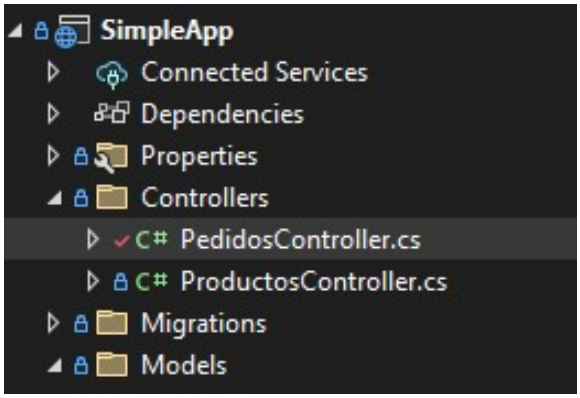
Una vez que la migración haya terminado, aplícalo a la base de datos con el siguiente comando.

PowerShell

```
dotnet ef database update
```

13) Creando servicios con MVC

Ahora crearemos la carpeta de **Controllers** y crearemos el **ProductosController.cs y PedidosController.cs**
Y pondremos el siguiente código en sus respectivos archivos.



PedidosController.cs

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using SimpleApp.Models;
using System.Collections.Generic;

namespace SimpleApp.Controllers
{
    [Route("api/[controller]")]
    public class PedidosController : Controller
    {
        private DataContext Context;
        public PedidosController(DataContext PContext)
        {
            Context = PContext;
        }
        [HttpGet]
        public IEnumerable<Pedido> GetPedidos()
        {
            IEnumerable<Pedido> Pedidos = Context.Pedidos;

            return Context.Pedidos;
        }

        [HttpGet("{id}")]
        public Pedido? GetPedido(long id)
        {
            return Context.Pedidos.Find(id);
        }

        [HttpGet("{id}/{detalle}")]
        public Pedido? GetPedido(long id, string detalle)
        {
            Pedido? Pedido = Context.Pedidos.Find(id);

            var TempProductos = (from Pedidos     in Context.Pedidos
                                 join PedidoProductos
                                 join Productos
                                 where Pedidos.PedidoId ==
                                 select new Producto
                                 {
                                     ProductoId = Productos
                                     Nombre     = Productos
                                     Precio     = Productos
```

ProductosController.cs

```
45                                                  });
46              if(TempProductos.Count() != 0)
47                  Pedido.Productos = TempProductos;
48              return Pedido;
```

C#

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

namespace SimpleApp.Controllers
{
    [Route("api/[controller]")]
    public class ProductosController : Controller
    {
        private DataContext Context;
        public ProductosController(DataContext PContext)
        {
            Context = PContext;
        }
        [HttpGet]
        public IEnumerable<Producto> GetProductos()
        {
            return Context.Productos;
        }

        [HttpGet("{id}")]
        public Producto? GetProducto(long id)
        {
            return Context.Productos.Find(id);
        }

        [HttpPost]
        public void SetProducto([FromBody] Producto P)
        {
            Context.Productos.Add(P);
            Context.SaveChanges();
        }

        [HttpPut]
        public void UpdateProducto([FromBody] Producto P)
        {
            Context.Productos.Update(P);
            Context.SaveChanges();
        }

        [HttpDelete("{id}")]
        public void DeleteProducto(long id)
        {
            Context.Productos.Remove(new Producto() { ProductoId = id});
```

```
            Context.SaveChanges();
        }
    }
}
```

Con esto finalizamos el proyecto

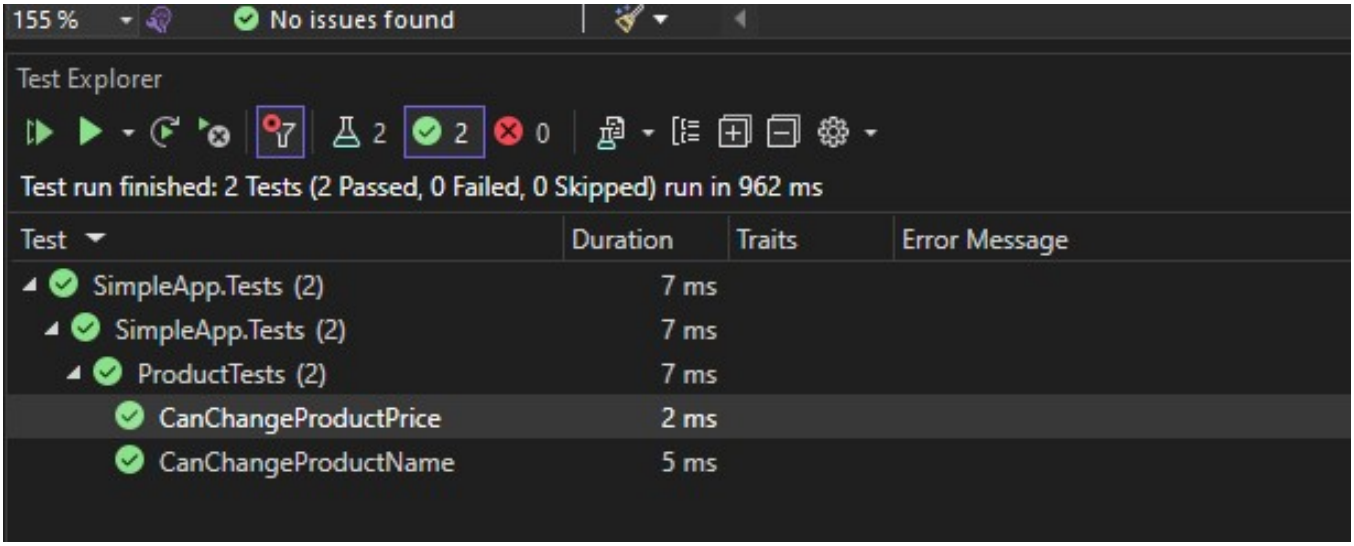
14) Añadiendo el Unit Test producto

Crearemos un archivo llamado **ProductoTests.cs** en la aplicación **SimpleApp.tests.** Para hacer pruebas en nuestra clase Producto.
A continuación el código.

C#

```
using SimpleApp.Models;
using Xunit;
namespace SimpleApp.Tests
{
    public class ProductTests
    {
        [Fact]
        public void CanChangeProductName()
        {
            // Arrange
            var     p = new Product { Name = "Test", Price = 100M };
```

Con esto podemos hacer pruebas de nuestra clase en **Visual Studio**, dando a **Test** → **Test Explorer** y dando clic a **Run All Test**.


155 %
No issues found
Test Explorer
2
2
0
Test run finished: 2 Tests (2 Passed, 0 Failed, 0 Skipped) run in 962 ms
Test
Duration
Traits
Error Message
SimpleApp.Tests (2)
7 ms
SimpleApp.Tests (2)
7 ms
ProductTests (2)
7 ms
CanChangeProductPrice
2 ms
CanChangeProductName
5 ms