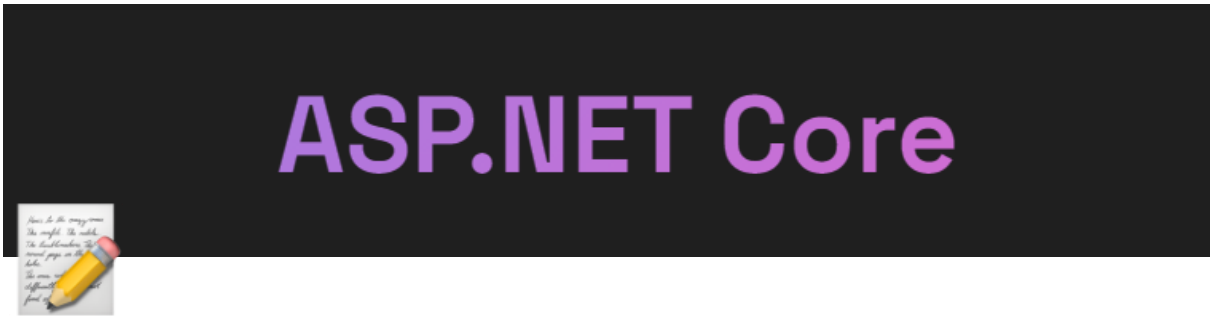


Prueba Técnica ASP.NET Core

Owner	Ⓡ Ruben HQ
Tags	Guides and Processes Testing

1. Creación de un API REST simple

Para la creación de una API en ASP.NET Core, se optó por un proyecto de tipo **ASP.NET Core Web API**. Dentro de este, se organizó una carpeta llamada **Models** que contiene la clase Producto.cs.

En la carpeta **Controllers**, se añadió un controlador llamado ControladorProducto.cs con el objetivo de implementar los endpoints necesarios para la gestión de productos. Además, se añadió una lista de productos de ejemplo para facilitar la interacción y prueba de los siguientes endpoints:

- Obtener todos los productos.

```
// Obtener todos los productos
[HttpGet]
public ActionResult<IEnumerable<Producto>> GetProductos()
{
    return productos;
}
```

- Obtener un producto por su ID

```
// Obtener un producto por su ID
[HttpGet("{id}")]
public ActionResult<Producto> GetProducto(int id)
{
    var producto = productos.FirstOrDefault(p => p.Id == id);
    if (producto == null)
        return NotFound();
    return producto;
}
```

- Agregar un nuevo producto.

```
// Agregar un nuevo producto
[HttpPost]
public ActionResult<Producto> AddProducto(Producto producto)
{
    producto.Id = productos.Max(p => p.Id) + 1; // Generar un nuevo ID
    productos.Add(producto);
    return CreatedAtAction(nameof(GetProducto), new { id = producto.Id }, producto);
}
```

- Actualizar un producto existente.

```
// Actualizar un producto existente
[HttpPut("{id}")]
public ActionResult UpdateProducto(int id, Producto productoActualizado)
{
    var producto = productos.FirstOrDefault(p => p.Id == id);
    if (producto == null)
        return NotFound();

    producto.Nombre = productoActualizado.Nombre;
    producto.Precio = productoActualizado.Precio;
    producto.CantidadStock = productoActualizado.CantidadStock;
    return NoContent();
}
```

💡 Nota: `ActionResult` hace posible que el método devuelva códigos de estado HTTP específicos. Por ejemplo, si una operación es exitosa, puedes devolver un **200 OK**, y si hay algún error, puedes devolver un **404 Not Found**, **500 Internal Server Error**, etc. Esto es útil para comunicar el estado de la solicitud al cliente de manera precisa.

💡 Nota:
En ASP.NET Core, las anotaciones `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, y `[HttpDelete]` se utilizan para indicar el tipo de solicitud HTTP que un método del controlador acepta.

`[HttpGet]` se utiliza para **leer o recuperar datos**.

`[HttpPost]` se utiliza para **crear un nuevo recurso** en el servidor.

`[HttpPut]` se utiliza para **actualizar un recurso existente**.

`[HttpDelete]` se utiliza para **eliminar un recurso existente** en el servidor.

2. Manejo de autenticación y autorización

Un JSON Web Token (JWT) define una forma segura y compacta de transmitir información entre dos entidades en forma de un objeto JSON.

Esta información puede ser verificada y es confiable ya que está firmada digitalmente.

Cuando un usuario ha sido autenticado, el servicio deberá regresar un JSON Web Token para ser usado como sus credenciales. Dado que esto es usado para autorizar el usuario, debes de considerar cuidar muy bien donde guardas el token, y eliminarlo lo más pronto posible si ya no se requiere.

Para esta prueba se optó por un proyecto de tipo **ASP.NET Core Web API**. Luego se instalaron los siguientes paquetes:

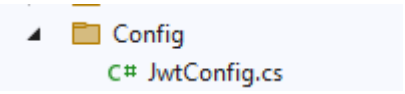
`Microsoft.AspNetCore.Authentication.JwtBearer`

`Microsoft.IdentityModel.Tokens`

Se creo una carpeta Models y en ella un archivo llamado Usuario.cs, este archivo contiene la clase Usuario con los campos id, nombre del usuario y contrasena.

```
namespace ParteDos.Models
{
    public class Usuario
    {
        public int Id { get; set; }
        public string NombreUsuario { get; set; } = string.Empty;
        public string Contrasena { get; set; } = string.Empty;
    }
}
```

Para configurar la generación del Token JWT se creó una nueva clase JwtConfig.cs en una carpeta Config. con la finalidad de almacenar la configuración específica del JWT permitiendo que los parámetros de autenticación como el token se manejen de forma segura y fácil de modificar.



```
namespace ParteDos.Config
{
    public class JwtConfig
    {
        public string token { get; set; } = string.Empty;

    }
}
```

En el archivo que viene por defecto appsettings.json se agregó una clave para firmar los tokens ya que es fundamental para la seguridad de la autenticación basada en JWT. Esta clave se usa para firmar y validar los tokens JWT, asegurando que el servidor puede confirmar la autenticidad de cada token emitido y que los tokens generados por el servidor son válidos.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "JwtConfig": {
    "token": "tokendeseguridad123456789"
  }
}
```

Para configurar el servicio de autenticación en el archivo Program.cs se agregaron las configuraciones para JWT en builder.Services.

```
// Configuración de JWT
builder.Services.Configure<JwtConfig>(builder.Configuration.GetSection("JwtConfig"));
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    var key = Encoding.ASCII.GetBytes(builder.Configuration["JwtConfig:token"]);
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

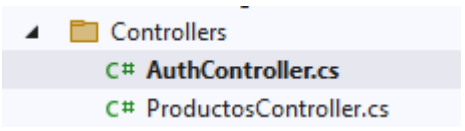
Para el Registro e Inicio de Sesión, en la carpeta Controllers, se creó un archivo AuthController.cs y se agregaron los endpoints para el registro e inicio de sesión.

```
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController : ControllerBase
    {
        private static List<Usuario> usuarios = new List<Usuario>();
        private readonly JwtConfig _jwtConfig;

        public AuthController(IOptions<JwtConfig> jwtConfig)
        {
            _jwtConfig = jwtConfig.Value;
        }

        // Registro de usuario
        [HttpPost("registro")]
        public IActionResult Registro([FromBody] Usuario usuario)
        {
            if (usuarios.Any(u => u.NombreUsuario == usuario.NombreUsuario))
                return BadRequest("El usuario ya existe.");

            usuario.Id = usuarios.Count + 1;
            usuarios.Add(usuario);
            return Ok("Usuario registrado con éxito.");
        }
```



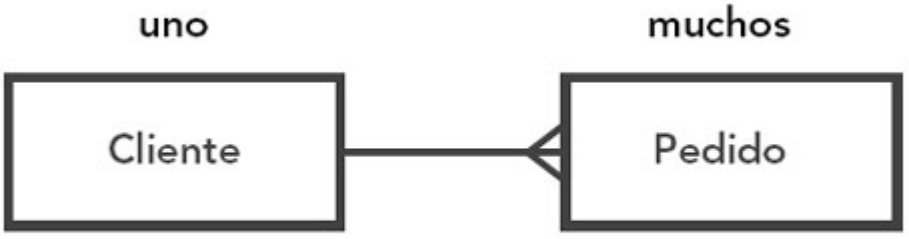
De último para proteger las rutas de la API, en la carpeta Controllers se encuentra el archivo ProductosController, y es ahi donde se aplica el atributo [Authorize] en la clase o en los métodos que se desea proteger.

```
namespace ParteDos.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    [Authorize] // Protege todas las rutas de este controlador
    public class ProductosController : ControllerBase
    {
        private static List<Producto> productos = new List<Producto>
        {
            new Producto { Id = 1, Nombre = "Leche Nido", Precio = 45.0m, CantidadStock = 10 },
            new Producto { Id = 2, Nombre = "Nescafe", Precio = 30.0m, CantidadStock = 5 },
            new Producto { Id = 3, Nombre = "Pan Bimbo", Precio = 25.0m, CantidadStock = 20 },
            new Producto { Id = 4, Nombre = "Jugo del Frutal", Precio = 15.0m, CantidadStock = 4 },
            new Producto { Id = 5, Nombre = "Mayonesa Anabelli", Precio = 35.0m, CantidadStock = 9 }
        };
```

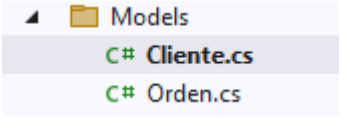
3. Consulta y manipulación de datos con Entity Framework Core

Para esta prueba se implementó una relación de 1 a muchos entre clientes y órdenes, donde:

- un cliente puede tener muchas órdenes
- cada orden tiene un identificador único, fecha de creación y el total



En la carpeta Models se crearon archivos, Cliente.cs y Orden.cs ambas son clases con sus respectivos atributos.



En la clase Cliente se establecieron los siguientes atributos.

```
public int ClienteId { get; set; }
public string Nombre { get; set; }

public ICollection<Orden> Ordenes { get; set; }
```

Este último se creó para indicarle a Entity Framework que la propiedad Ordenes es una relación de uno a muchos.

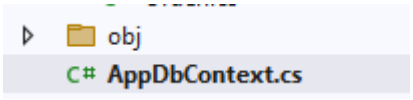
Por otra parte la clase Orden se crearía con los siguientes atributos.

```
public class Orden {
    public int OrdenId { get; set; }
    public DateTime Fecha { get; set; }
    public decimal Total { get; set; }

    //para indicar que es llave foranea
    public int ClienteId { get; set; }

    //para navegar a traves de la relacion
    public Cliente Cliente { get; set; }
}
```

En cuanto a la conexión a la base de datos se creó un archivo AppDbContext.cs y se crearía una clase llamada AppDbContext



```
namespace ParteTres
{
    public class AppDbContext : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Server=DESKTOP-0FGOU17;Database=pruebaTecnicaPrinter;Trusted_Connectio
        }

        public DbSet<Cliente> Clientes { get; set; }
        public DbSet<Orden> Ordenes { get; set; }
    }


}
```

La clase hereda de
`DbContext`, que es el componente central de EF Core para interactuar con la base de datos.

El método
`OnConfiguring` se utiliza para configurar las opciones de conexión de la base de datos.

`optionsBuilder.UseSqlServer(...)`: Especifica que EF Core debe usar SQL Server como el proveedor de base de datos.

La cadena de conexión

```
"Server=DESKTOP-0FGOU17;Database=pruebaTecnicaPrinter;Trust
ed_Connection=True;"
```

define cómo conectarse a la base de datos. Aquí:

`Server=DESKTOP-0FGOU17`: El nombre del servidor de SQL Server.

`Database=pruebaTecnicaPrinter`: El nombre de la base de datos a la que se conectará.

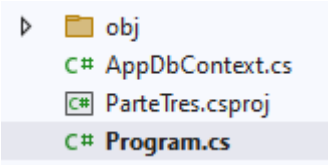
`Trusted_Connection=True`: Usa la autenticación integrada de Windows para conectarse.

`DbSet<T>` representa una tabla en la base de datos, y cada entidad `DbSet` actúa como un conjunto de registros (filas) de esa tabla.

`DbSet<Cliente> Clientes`: Define una tabla `Clientes` que almacena objetos de tipo `Cliente`.

`DbSet<Orden> Ordenes`: Define una tabla `Ordenes` que almacena objetos de tipo `Orden`.

Respecto a la consulta en LINQ fue creada en el archivo Program.cs




```
using Microsoft.EntityFrameworkCore;
using ParteTres;
using ParteTres.Models;

using (var context = new AppDbContext())
{
    var cliente = new Cliente { Nombre = "Luis Zepeda" };
    context.Clientes.Add(cliente);
    await context.SaveChangesAsync();

    var orden = new Orden { Fecha = DateTime.Now, Total = 150.00m, ClienteId = cliente.ClienteId };
    context.Ordenes.Add(orden);
    await context.SaveChangesAsync();

    // Obtener el total de las órdenes para un cliente específico
    var totalOrdenes = await context.Ordenes
        .Where(o => o.ClienteId == cliente.ClienteId)
        .SumAsync(o => o.Total);

    Console.WriteLine($"Total de órdenes para el cliente {cliente.Nombre}: {totalOrdenes}");
```

Nota: Una *consulta* es una expresión que recupera datos de un origen de datos.

LINQ permite escribir consultas en un estilo declarativo, lo cual mejora la legibilidad y hace que el código sea más fácil de entender.

LINQ se puede utilizar no solo con bases de datos (a través de LINQ to SQL o LINQ to Entities en Entity Framework) sino también con colecciones en memoria (arrays, listas), XML, y servicios. Esto significa que se puede usar la misma sintaxis y estilo de consulta para diferentes tipos de datos.

`using` crea una instancia de `AppDbContext` y garantiza que los recursos asociados se liberen adecuadamente al finalizar el bloque. Al cerrar el contexto, se aseguran que las conexiones de base de datos se cierren correctamente.

`new Cliente { Nombre = "Luis Zepeda" }`: Crea una nueva instancia de la entidad `Cliente` con el nombre "Luis Zepeda".

`context.Clientes.Add(cliente);`: Agrega la instancia del cliente a la tabla `Clientes` en la base de datos.

`await context.SaveChangesAsync();`: Guarda los cambios en la base de datos de forma asincrónica, insertando el nuevo cliente y generando un `ClienteId` para el cliente.

`new Orden { Fecha = DateTime.Now, Total = 150.00m, ClienteId = cliente.ClienteId }`: Crea una nueva instancia de la entidad `Orden`, asignando la fecha actual, un total de `150.00`, y asociándola al cliente recién creado usando `ClienteId`.

`context.Ordenes.Add(orden);`: Agrega la orden a la tabla `Ordenes` en la base de datos.

`await context.SaveChangesAsync();`: Guarda los cambios, insertando la orden en la base de datos.

`context.Ordenes.Where(...)`: Filtra las órdenes en la base de datos para obtener solo aquellas donde `ClienteId` coincide con el ID del cliente creado (`cliente.ClienteId`).

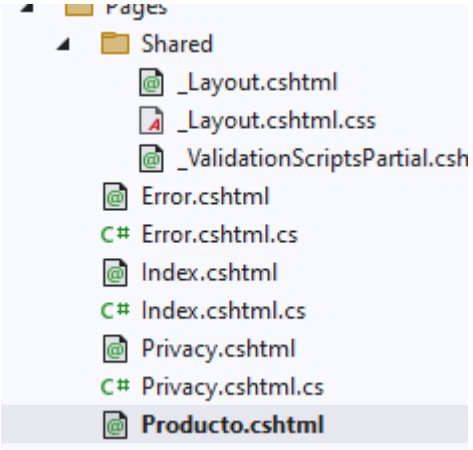
`SumAsync(o => o.Total)`: Suma el valor `Total` de todas las órdenes que pertenecen a ese cliente, obteniendo el total de órdenes.

`await`: Se realiza de forma asincrónica para optimizar el rendimiento y evitar bloquear el hilo principal de ejecución.

4. Diseño de una página web responsiva con Razor Pages

Para elaborar el diseño de la página web se optó por un proyecto de tipo **web ASP.NET Core (Razor Pages)**.

Se creó el archivo Producto en la carpeta Pages, en este archivo diseñó el formulario en formato html utilizando bootstrap para que fuera responsive.




```
@page
@model ParteCuatro.Pages.ProductoModel
@{
}

<div class="row">
    <div class="form-control form-control-lg">
        <h2 class="text-center mb-5">Registro de Producto</h2>
        <form method="post">
            <div class="row mb-3">
                <label class="col-sm-4 col-form-label">Id Producto</label>
                <div class="col-sm-8">
                    <input class="form-control" asp-for="id">
                    <span asp-validation-for="id" class="text-danger"></span>
                </div>

            </div>

            <div class="row mb-3">
                <label class="col-sm-4 col-form-label">Nombre Producto</label>
                <div class="col-sm-8">
                    <input class="form-control" asp-for="nombre">
                    <span asp-validation-for="nombre" class="text-danger"></span>
```

El archivo Producto.cshtml.cs contiene:

```
namespace ParteCuatro.Pages
{
    public class ProductoModel : PageModel
    {
        [BindProperty]
        [Required (ErrorMessage = "El campo ID es requerido")]
        public int id { get; set; } = 0;


        [BindProperty]
        [Required(ErrorMessage = "El campo Nombre es requerido")]
        public string nombre { get; set; } = "";


        [BindProperty]
        [Required(ErrorMessage = "El campo Precio es requerido")]
        public double precio { get; set; } = 0;
```

La clase `ProductoModel` es utilizada para gestionar los datos de un formulario de productos, permitiendo la vinculación de propiedades y la validación de datos en el front-end.

La clase `ProductoModel` hereda de `PageModel`, que es una clase base para modelos de páginas en Razor Pages. Esto permite que `ProductoModel` gestione la lógica de la página y los datos asociados a los elementos del formulario.

`[BindProperty]`: Permite la vinculación de datos entre la propiedad `id` y los campos del formulario en el front-end.

`[Required]`: Aplica una validación obligatoria. Si el usuario deja el campo `id` vacío, mostrará el mensaje de error `"El campo ID es requerido"`.

En el archivo Program.cs se escribió la siguiente instrucción para que la aplicación inicializara con el debido formulario.

```
app.MapGet("/", () => Results.Redirect("/Producto"));
```

5. Depuración y manejo de errores

En una aplicación ASP.NET Core, es importante manejar los errores globalmente para proporcionar una respuesta adecuada en caso de fallos, sin exponer detalles sensibles de la aplicación.

Para elaborar esta prueba se optó por un proyecto de tipo **web ASP.NET Core (Razor Pages)**.

Se creó una carpeta llamada Middlewares y en ella se creó un archivo llamado ErrorHandlingMiddleware el objetivo fue crear una clase que actúa como un middleware personalizado que captura cualquier excepción que ocurra en los middlewares o en la aplicación, y la gestiona de forma global.

```
public class ErrorHandlingMiddleware
{
    private readonly RequestDelegate _next; // Declaración de _next
    private readonly ILogger<ErrorHandlingMiddleware> _logger;

    public ErrorHandlingMiddleware(RequestDelegate next, ILogger<ErrorHandlingMiddleware> logger)
    {
        _next = next; // Inicialización de _next
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context); // Llamar al siguiente middleware en la cadena
        }
        catch (Exception ex)
        {
            await HandleExceptionAsync(context, ex);
        }
```

`RequestDelegate _next`: Representa el siguiente middleware en la cadena. Este delegado se usa para pasar la solicitud al siguiente middleware una vez que se ha completado la lógica de este middleware.

`ILogger<ErrorHandlingMiddleware> _logger`: Permite registrar mensajes de error o eventos. Aquí, se utiliza para registrar cualquier excepción que ocurra en el proceso.

El constructor recibe el `RequestDelegate` y el `ILogger` y los asigna a las propiedades privadas. Esto permite que el middleware se integre en la cadena de middlewares y registre errores.

`try`: Intenta ejecutar el siguiente middleware en la cadena con `await _next(context);`.

`catch (Exception ex)`: Si ocurre una excepción en la cadena de middlewares, esta se captura y se envía al método `HandleExceptionAsync` para manejarla de forma centralizada.

Se utilizó el archivo Program.cs para configurar el middleware de manejo de errores.

```
// Configurar el middleware de manejo de errores.
app.UseMiddleware<ErrorHandlingMiddleware>();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Error"); // Esto redirige a una página de error personalizada.
    app.UseHsts();
}
```

`app.UseMiddleware<ErrorHandlingMiddleware>();` agrega el middleware `ErrorHandlingMiddleware` a la canalización de solicitudes. Este middleware intercepta las solicitudes y, en caso de una excepción, gestiona el error de forma global y centralizada.

Al utilizar `ErrorHandlingMiddleware`, cualquier excepción lanzada en la aplicación será capturada, registrada y respondida con un mensaje genérico (configurado previamente en `ErrorHandlingMiddleware`), lo cual evita la exposición de información técnica sensible a los usuarios.

`app.Environment.IsDevelopment()` verifica si la aplicación se está ejecutando en el entorno de desarrollo.

Si está en desarrollo:

`app.UseDeveloperExceptionPage();` habilita la página de excepciones para desarrolladores, la cual muestra información detallada sobre el error. Esto es útil para depurar durante el desarrollo, ya que muestra la pila de llamadas y detalles de la excepción.

Si está en producción:

`app.UseExceptionHandler("/Error");` configura un controlador de excepciones que redirige a una página de error personalizada en `/Error`. Así, en producción, el usuario es dirigido a una página específica de error en lugar de ver los detalles de la excepción.

`app.UseHsts();` habilita el HTTP Strict Transport Security (HSTS), una política de seguridad que fuerza a los navegadores a conectarse a través de HTTPS para todas las solicitudes futuras. Esto solo se activa en producción para mejorar la seguridad de la aplicación.

También se creó un archivo llamado Test.cshtml.cs con la finalidad de que se lanzara una excepcion al momento de iniciar la aplicación.

```
public class TestModel : PageModel
{
    public IActionResult OnGet()
    {
        throw new Exception("Esto es una excepcion!"); // Lanzar la excepción
    }
}
```

La clase
`TestModel` hereda de `PageModel`, lo cual la convierte en una clase modelo para

una página Razor en ASP.NET Core. Al heredar de `PageModel`, esta clase puede manejar las solicitudes HTTP y las interacciones con la vista de Razor Pages.

`OnGet` es un método de controlador de páginas en Razor Pages que se ejecuta en respuesta a una solicitud GET. Cada vez que un usuario accede a la página asociada, el método `OnGet` se llama automáticamente.

El tipo de retorno es `IActionResult`, lo cual permite devolver varios tipos de respuestas HTTP (por ejemplo, vistas, redirecciones o resultados JSON).

`throw new Exception("Esto es una excepcion!");` lanza una excepción con el mensaje "Esto es una excepcion!". En este caso, se lanza intencionalmente para simular un error y comprobar cómo la aplicación maneja las excepciones.