Taller de Lenguajes II – 2025
Programador Universitario / Licenciatura en informática / Ingeniería en Informática
TP Nro 11

CONTENIDOS

- Log
- Try-catch

INTRODUCCIÓN

Seguiremos trabajando con el Repositorio del Trabajo Práctico anterior.
En esta nueva iteración del desarrollo se incorporarán mecanismos de **logueo** y **control de excepciones**, con el objetivo de hacer el sistema más robusto y confiable. Además, se configurará la cadena de conexión en el archivo **appsettings.json**, para que pueda ser inyectada en los repositorios mediante inyección de dependencia

1. **Para realizar el logueo** se deberá incluir logueo en el endpoint de control de acceso, contemplando los siguientes casos:
   - **Acceso exitoso:** Registrar un log de tipo **Information** mostrando por consola el siguiente mensaje: “El usuario *UsuarioLogueado* ingresó correctamente”.
   - **Acceso rechazado:** Registrar un log de tipo **Warning** mostrando por consola el siguiente mensaje: “Intento de acceso inválido + Usuario: *UsuarioLogueado* + Clave ingresada: *Clave*”.
   - **Errores en ejecución:** En los bloques try-catch, incluir logueo de tipo **Error**, mostrando por consola el detalle de la excepción serializada utilizando el método ToString(). (Ver Ejemplo 1).
2. **Para el control de excepciones**: Se deberá agregar un bloque try-catch en cada endpoint del sistema con el fin de capturar cualquier error que pueda producirse durante su ejecución. En caso de que ocurra una excepción: (Ver Ejemplo 1).
   - Registrar el error mediante el sistema de logueo.
   - Retornar al usuario una página de error apropiada.

Ejemplo 1 - Manejo de excepciones en endpoints:

```
[HttpPost]
 public IActionResult CrearProducto(CrearProductoViewModel nuevoProducto){
     try{
         var productoNuevo = new Producto(nuevoProductoVM);
         productoRepo.CrearProducto(productoNuevo);
         return RedirectToAction("ListarProductos");
     }
     catch (Exception ex)
     {
         logger.LogError(ex.ToString());
          return BadRequest();
     }
 }
```

Para cada método de los repositorios del sistema, implemente un sistema para enviar excepciones en casos donde la consulta no resulta exitosa.

Ejemplo: Si se envía un id de un producto que no existe en la base de datos, debería enviar una excepción informando este error. Ver Ejemplo 2

Ejemplo 2 - Disparar una excepción en caso de que una consulta resulte vacía.

```
public Producto GetProducto(int idProducto){
       Producto producto =null;
       using(var connection = new SQLiteConnection(_connectionString))
       {
           connection.Open();
           string queryString = @"SELECT * FROM Productos WHERE id = @idProducto";
           var command = new SQLiteCommand(queryString, connection);
           command.Parameters.Add(new SQLiteParameter("@idProducto", idProducto));
           using(var reader = command.ExecuteReader())
           {
               if(reader.Read()){
                   producto.Id = Convert.ToInt32(reader["idProducto"]);
                   producto.Nombre = reader["Descripcion"].ToString();
                   producto.Precio = reader["Precio"].ToString();
               }
           }
           connection.Close();
       }
       if (producto ==null)
          throw new Exception("Producto inexistente");
       return tablero;
   }
```

3. **Vamos a incorporar un último refinamiento** al sistema que nos permitirá controlar la cadena de conexión de la aplicación desde un archivo externo. Para ello, utilizaremos el archivo **appsettings.json**, que ya se encuentra en la raíz del proyecto, e inyectamos la cadena de conexión recuperada directamente en los repositorios.:
   a. Agregue la entrada correspondiente en el archivo **appsettings.json** y ajustela con los valores adecuados según la base de datos que esté utilizando, de la siguiente manera:

```
{
 [...] // otras configuraciones
 "ConnectionStrings": {
   "SqliteConexion": "Data Source=db/Tienda.db;"
 }
}
```

b. Inyecte la cadena de conexión en los repositorios recuperando su valor desde el archivo **appsettings.json** a través del objeto **builder**. Para ello, agregue el siguiente código en el archivo **Program.cs:**

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllersWithViews();
// líneas de código a incorporar
var CadenaDeConexion = builder.Configuration.GetConnectionString(
"SqliteConexion")!.ToString();
builder.Services.AddSingleton<string>(CadenaDeConexion);

// Aquí se realiza la inyección de los repositorios
// [...]
//
var app = builder.Build();
```

Como se vió anteriormente, **builder.Services** es el contenedor de servicios de inyección de dependencias que utiliza ASP .Net para registrar y resolver las dependencias de la aplicación.

AddSingleton<string>(CadenaDeConexion)
Registra el valor de **CadenaDeConexion** como un servicio *Singleton* del tipo `string`. Esto implica que la cadena de conexión tendrá una única instancia compartida en toda la aplicación y cada clase que solicite un `string` en su constructor recibirá esta misma instancia.

c. En los repositorios que acceden a la base de datos, la cadena de conexión se recibe a través del constructor. Para ello, el repositorio define un campo privado donde se almacena dicha cadena, la cual es inyectada al momento de crear la instancia mediante la configuración realizada en **Program.cs**.

De esta forma, el repositorio queda desacoplado de la configuración y puede utilizar la cadena de conexión sin conocer su origen, facilitando el mantenimiento y el cambio de entornos.

```
public class PresupuestosRepository: IPresupuestosRepository
{
    private readonly string _ConnectionString;
    public PresupuestosRepository(string ConnectionString)
    {
        _ConnectionString=ConnectionString;
```

```
    }

[...]
```