Taller de Lenguajes II – 2024
Programador Universitario / Licenciatura en informática / Ingeniería en Informática
TP Nro 9

CONTENIDOS

- Logueo
- Try-catch

INTRODUCCIÓN

Seguiremos trabajando con el Repositorio del Trabajo Práctico anterior.

En esta nueva iteración de trabajo, que venimos desarrollando vamos a agregar controles de logueo y control de excepciones, con esto seguimos haciendo un desarrollo más robusto. Además se agregará la cadena de conexión al archivo appsettings.json para que se inyecten en los repositorios.

1. Logueo:
    1. Incluir en el endpoint de control de acceso:
        a. Cuando el acceso fue exitoso, muestre por consola logueo de tipo info la siguiente leyenda “el usuario + UsuarioLogueado “ + ingreso correctamente
        b. Cuando el acceso fue rechazado, muestre por consola logueo de tipo warning la siguiente leyenda “Intento de acceso invalido - Usuario:” +UsuarioLogueado+” Clava ingresada: ” + Clave
    2. En los bloques Try-catch incluya logueo del tipo error, muestre por consola la serializando el error utilizando el comando toString(). Ver Ejemplo 1

2. Control de excepciones:
    a. Agregue un bloque Try-Catch en cada endpoint del sistema para garantizar que cualquier error que pudiera ocurrir durante la ejecución de estos pueda ser capturada. En caso de ocurrir una excepción, loguee el error , y envíe al usuario una página de error. Ver Ejemplo 1

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();
     }
 }
```

b. 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. Inyección de Cadena de conexión:
    a. Agregue en los archivos appsettings.json. la siguiente línea:

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

b. Inyecte la conexión en los repositorios recuperando la información del archivo appsettings.json en el objeto builder(). Para eso agregue el siguiente código al 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();
```

Nota: Explicación de la línea de código
builder.Services.AddSingleton<string>(CadenaDeConexion);

`builder.Services`:

- Es el contenedor de servicios de inyección de dependencias que ASP.NET Core utiliza para registrar y resolver dependencias en la aplicación.

`.AddSingleton<string>(connectionString)`:

- Registra el valor de `connectionString` como un servicio Singleton para el tipo `string`.
- Esto significa que:
  - La cadena de conexión será una única instancia compartida en toda la aplicación.
  - Si alguna clase solicita un `string` en el constructor, esta instancia será proporcionada.

c. En los Repositorios que quiera inyectar la cadena de conexión el constructor tiene que quedar de la siguiente forma:

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