TEMA 11. CONSISTENCIA DE DATOS. SERIALIZACIÓN Y CONEXIÓN A BD MYSQL

1. Persistencia

Se define por persistencia en el mundo de la POO, como la capacidad que tienen los objetos de sobrevivir al proceso padre que los creo. La persistencia permite al programador almacenar, transferir y recuperar fácilmente el estado de los objetos.

¿Cómo Podemos Conseguir La Persistencia?

La forma más común de conseguirlo es mediante la serialización.

2. Serialización

La serialización es el proceso de convertir el estado de un objeto a un formato que se pueda almacenar o transportar. El complemento de la serialización es la deserialización, que convierte una secuencia a un objeto. Juntos, estos procesos permiten almacenar y transferir fácilmente datos.

.NET Framework ofrece dos tecnologías de serialización:

- La serialización binaria conserva la fidelidad de tipos, lo que resulta útil para conservar el estado de un objeto entre distintas llamadas a una aplicación. Puedes serializar un objeto en una secuencia, un disco, la memoria, a través de la red, etc. Por ejemplo, se puede compartir un objeto entre distintas aplicaciones si lo serializa en el Portapapeles.
- La serialización XML sólo serializa las propiedades públicas y los campos, y no conserva la fidelidad de tipos. Esto resulta útil cuando desea proporcionar o consumir datos sin restringir la aplicación que utiliza los datos. Como XML es un estándar abierto, es una opción atractiva para compartir datos a través del Web.

- La serialización a alguna notación de objetos estándar como JSON.

Aunque la mayoría de lenguajes ya la traen implementada. Si tuviéramos que definir nosotros un interfaz OO para serializar objetos. Podría ser algo parecido a esto...

```
public interface ISerializacion<T>
{
   void Serializa(T dato, Stream flujo);
   T Deserializa(Stream flujo);
}
```

A la hora de serializar una clase llamaríamos a su método Serializa y este a su vez a los Serializa de los objetos y tipos que contenga, así sucesivamente. Muchos lenguajes como Java o C# solucionan la serialización de forma sencilla, ya que al serializar un objeto contenedor, este a su vez serializa mediante un mecanismo de reflexión y de

forma trasparente aquellas referencias a objetos que contiene. Lo mismo sucede al cargar o deserializar un objeto.

Durante este proceso, los campos público y privado del objeto y el nombre de la clase, incluido el ensamblado que contiene la clase, se convierten en una secuencia de bytes que, a continuación, se escribe en una secuencia de datos. Cuando, después, el objeto se deserializa, se crea una copia exacta del objeto original.

Para ello C# me ofrece marcar mis clases como serializables a través de **un atributo**.

Atributo en .NET → un atributo es una etiqueta de la sintaxis [nombre] que podremos aplicar a un ensamblado, clase, constructor, enumeración, campo, interfaz, método, etc... y que genera información en el ensamblado en forma de metadatos heredando de la clase Attribute.

Por ejemplo si queremos realizar una simple serialización binaria etiquetaremos la clase a serializar y todas las que contenga con el atributo ya definido **[Serializable]**, sobre la definición de la clase..

[serializable]

```
 [Serializable]
public class MiClase
{
    public int n1;
    public int n2;
    public String str;
}
```

Para posteriormente serializar el tipo deberemos utilizar un formateador, el más común es el IFormatter, que se utiliza de la siguiente manera:

```
IFormatter <nombreobjeto> = new BinaryFormatter();
<nombreobjeto>.Serialize(<medioalmacenamiento>, <objetoaserializar>);
```

En el siguiente ejemplo de código se muestra cómo serializar una instancia de esta clase en un archivo.

```
MiClase obj = new MiClase();
obj.n1 = 1;
obj.n2 = 24;
obj.str = "Some String";
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("MyFile.bin", FileMode.Create, FileAccess.Write);
formatter.Serialize(stream, obj);
stream.Close();
```

Este ejemplo utiliza un formateador binario para realizar la serialización. Todo lo que necesita es crear una instancia de la secuencia y el formateador que desee utilizar y, a continuación, llamar al método **Serialize** en el formateador.

Igual de sencillo resulta restaurar el objeto a su estado anterior. En primer lugar, cree una secuencia para leer y un formateador a continuación, indique el formateador que deserialice el objeto. Su generalización sería:

```
IFormatter <nombreobjeto> = new BinaryFormatter();
Objeto=<nombreobjeto>.Deserialize(<medioalmacenamiento>);
```

En el siguiente ejemplo de código se muestra cómo hacerlo.

```
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("MyFile.bin", FileMode.Open, FileAccess.Read);
MiClase obj = (MiClase) formatter.Deserialize(stream);
stream.Close();

// Here's the proof.
Console.WriteLine("n1: {0}", obj.n1);
Console.WriteLine("n2: {0}", obj.n2);
Console.WriteLine("str: {0}", obj.str);
```

Es importante tener en cuenta que el atributo **Serializable** no se puede heredar. Si deriva una nueva clase de MiClase, la nueva clase debe marcarse también con el atributo; de lo contrario, no se podrá serializar.

Una clase a menudo contiene campos que no se deben serializar. Por ejemplo campos específicos que almacenen datos confidenciales. Si no se excluye estos campos de la serialización, los datos que almacenan dichos campos estarán expuestos a cualquier código que tenga permiso de serialización. Para que un campo no se serialize, deberemos aplicarle el atributo [**NonSerialized].**

```
[Serializable]
public class MiClase
{
  public int n1;
  [NonSerialized] public int n2;
  public String str;
}
```

Ejemplo más completo de serialización:

```
[Serializable]
class Persona
{
  public string Nombre;
  public int Edad;
  public string NIF;
  public Persona() { }
  public Persona(string nombre, int edad, string nif)
  {
     Nombre = nombre;
     Edad = edad;
     NIF = nif;
  }
  void Cumpleaños()
  {
     Edad++;
```

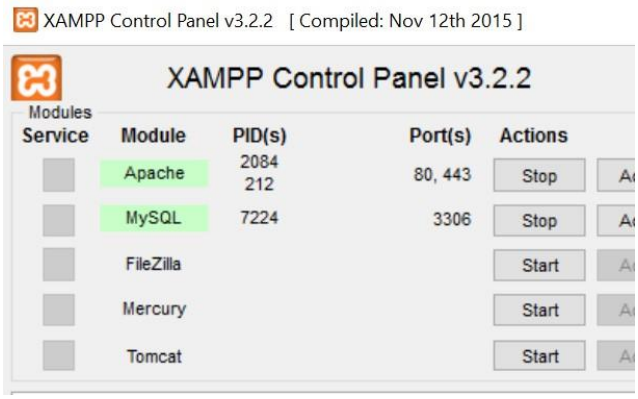
```
  }
 //Método que muestra la edad de la persona cuando pasen X años
 //utilizamos el operador this para llamar al método Cumpleaños
  void MostrarEdadFutura(int años)
  {
     for (int i = 0; i < años; i++) this.Cumpleaños();
     Console.WriteLine(Edad);
  }
}
class Program
{
  static void Serializar()
  {
    FileStream f = new FileStream(@"C:\datos.txt", FileMode.Append,FileAccess.Write);
    Persona p= new Persona("lulu", 10, "22112211L");
    IFormatter formatter = new BinaryFormatter();
    formatter.Serialize(f, p);
    f.Close();
  }
  static void Deserializar()
  {
    FileStream f = new FileStream(@"C:\datos.txt", FileMode.Open, FileAccess.Read);
    IFormatter formatter = new BinaryFormatter();
    Persona p;
    do
    {
      p = (Persona)formatter.Deserialize(f);
      Console.WriteLine(p.Nombre + " " + p.NIF + " " + p.Edad);
    } while (f.Position <f.Length);
    f.Close();
}
static void Main()
{
  Serializar();
  Deserializar();
  }
}
```

3. Conexión a BD MySQl desde C#

1.2. Que es una conexión a base de datos

Una conexión es como una clase de puente que realizamos desde nuestro lenguaje de programación hasta una base de datos, y se utiliza para acceder a ciertos recursos que nos provee el motor de base de datos, estos recursos entre otras cosas son: poder ejecutar sentencias SQL, hacer llamadas a procedimientos y llamadas a funciones.

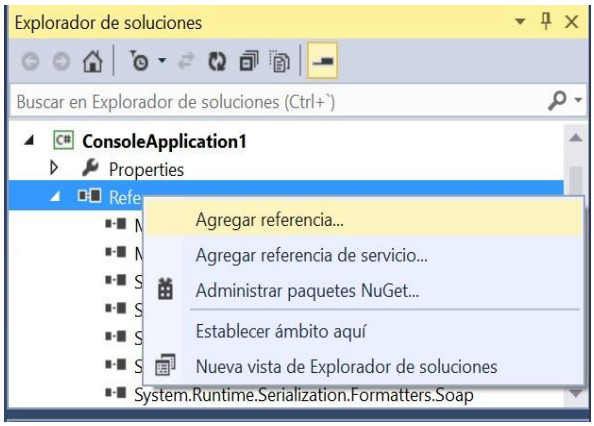
Como estamos trabajando con el servidor local de BD que provee XAMP, lógicamente tendremos que tener creada una BD con las tablas, usuario y demás elementos necesarios. Y también deberán estar activos los servicios Apache y MySql en el momento que queramos hacer la conexión.



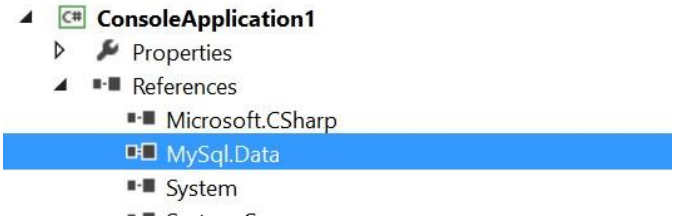
Para poder crear la conexión y posteriormente realizar todas las operaciones que necesitemos, tendremos que añadir una librería que nos aportará los métodos necesarios.

1.2. Referenciando al conector de mysql

El paso a seguir, es agregar una referencia de mysql a nuestro proyecto, para lograrlo, hacemos clic derecho sobre la carpeta referencias de nuestro proyecto y seleccionamos Agregar referencia:



Agregaremos la librería **MySql.Data.dll**, que se os ha pasado como recurso. Aunque se dispones de una referencia al API de este namespace en la siguiente URL https://dev.mysql.com/doc/dev/connector-net/6.10/html/N_MySql_Data_MySqlClient.htm Luego verificamos que la referencia se haya realizado correctamente:



Ah, no se nos puede olvidar referenciar nuestra dll desde código con:

```
using MySql.Data.MySqlClient;
```

Con eso, ya podemos utilizar todas las clases de acceso a datos, ofrecidas por mysql.

1.3. Crear string de conexión

Necesitaremos crear un string de conexión con los datos necesarios para conectarse a la BD, lógicamente esos datos deberán ser los mismos que pusimos en el momento de crear la BD. Y será necesario nombrarlos con un alias preestablecido. La manera correcta es la siguiente:

```
Server="";Port="";DataBase="";Uid="";Pwd="";
```

Dentro de las comillas va la respectiva información, en nuestro caso puntual:

```
Server=127.0.0.1;Port=3306;DataBase=prueba;Uid=root;Pwd=1234;
```

1.4. Abrir la conexión

Para abrir la conexión deberemos de crear un objeto de tipo MySqlConnection, asignar a su propiedad ConnectionString el string de conexión como hemos visto en el punto anterior y proceder a llamar al método Open. No olvidar añadir la referencia a MySql.Data.MySqlClient;

```
//abrir la conexion
MySqlConnection conexion = new MySqlConnection();
conexion.ConnectionString = "Server=127.0.0.1;Port=3306;DataBase=prueba;Uid=root;Pwd=1234;";
conexion.Open();
```

No deberemos abrir más de una conexión a la vez, ya que es una operación costosa temporalmente y podremos saturar a MySQL y rechazarnos la misma si tenemos muchas abiertas. Si es necesario, nos guardaremos o pasaremos el objeto con la conexión a aquellos métodos que lo necesiten. Si vas a realizar diferentes consultas de modificación seguidas como insert, update o delete. Es importante hacerlas usando la misma conexión.

Para probar los ejemplos que aparecerán en el resto del documento, deberás crearte una Base de Datos con una sola tabla que se llame **tabla1** con tres campos que se llaman a, b, c. El campo a y c son de tipo Int y el campo b de tipo Varchar(25).

1.5. Ejecutar una sentencia SQL

Una vez abierta la conexión ya podremos trabajar con nuestra BD. Una de las cosas que podemos hacer es ejecutar sentencias SQL de cualquier tipo, tanto para crear tablas, como para eliminarlas o modificar datos y como no también para consultar o modificar el contenido de las tablas. Para poder hacer todo esto tendremos que crearnos un objeto del tipo MySqlComman al que le asignaremos la conexión, indicaremos la sentencia select en la propiedad commandText y la ejecutaremos.

```
MySqlCommand cmd = new MySqlCommand();
cmd.Connection = conexion;
cmd.CommandText = "insert into tabla1(a,b,c) values('10','hola','111');";
cmd.ExecuteReader();
cmd.Connection.Close();
```

Con este procedimiento tan sencillo ya habremos insertado un registro en la base de datos.

Si quisiéramos leer datos de la BD, necesitaremos usar una clase más que recibirá el resultado de la consulta para posteriormente tratarlo, para esto se utiliza un objeto de tipo MySqlDataReader donde se guardará el resultado y al que podemos acceder para extraer los valores de los campos, normalmente por posición. A este tipo de objetos se les llama cursores y los veréis más ampliamente en segundo:

```
MySqlCommand cmd = new MySqlCommand();
cmd.Connection = conexion;
cmd.CommandText ="SELECT * FROM tabla1";

MySqlDataReader consultar;
consultar = cmd.ExecuteReader();
while (consultar.Read())
{
    int a = consultar.GetInt32(0);
    string b = consultar.GetString(1);
    int c = consultar.GetInt32(2);
    Console.WriteLine("Resultado:\n" + a + " " + b + " " + c + "\n");
}
cmd.Connection.Close();
```

Para evitar excepciones deberemos tener en cuenta el tipo de dato guardado en el campo de la tabla y la posición que ocupa. Cada vez que se realiza un consultar.Read() se pasará al registro siguiente de la tabla, devolverá false cuando no existan más registros.

1.6. Realizar la llamada a un procedimiento

En el caso de las llamadas al procedimiento, obviamente, deberemos tener creado y añadido el procedimiento a la BD antes de la invocación desde nuestro programa, en caso contrario se producirá una excepción. Tendremos que saber el nombre del procedimiento, y dependiendo de la forma en que creemos nuestro código, necesitaremos saber también o el nombre de los parámetros que queremos pasar o la posición que ocupan, en los dos casos deberemos saber de que tipo son. Primero vamos a ver un ejemplo suponiendo que sabemos el nombre de los parámetros:

```
public static void InvocarProcedimiento(int pValor1, string pValor2, int pValor3)
{
        try
        {
            //Abrir conexión
            MySqlConnection conexion = new MySqlConnection();
            if (conexion.State != ConnectionState.Open)
            {
                conexion.ConnectionString = "Server=127.0.0.1;Port=3306;DataBase=prueba;Uid=xusa;Pwd=1234;";
                conexion.Open();

                MySqlCommand cmd = new MySqlCommand();
                cmd.Connection = conexion;
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.CommandText = "insertar"; //nombre del procedimiento

                //asignar argumentos del procedimiento por nombre
                cmd.Parameters.AddWithValue("a", pValor1);
                cmd.Parameters.AddWithValue("b", pValor2);
                cmd.Parameters.AddWithValue("c", pValor3);
                //ejecutar el procedimiento
                cmd.ExecuteNonQuery();
                cmd.Connection.Close();
            }
        }
        catch (Exception ex)
```

Como podemos ver en el código, toda la parte referente a la apertura y conexión de BD es exactamente igual. Como no va a ser una ejecución de sentencia SQL deberemos indicar que tipo es en la propiedad CommandType, en este caso StoreProcedure y el nombre del procedimiento en la propiedad CommandText. Añadiremos los parámetros usando la técnica de clave/valor donde la clave será el nombre del argumento en el procedimiento y valor el valor que le queramos dar. Para ejecutar el comando, en este caso se usará ExecuteNonQuery.

Ejemplo del procedimiento referente a este ejemplo:

```
DELIMITER $$
create procedure insertar(a int, b varchar(25), c int)
begin
  insert into tabla1 values (a, b, c);
end;
$$
DELIMITER ;
```

Como se ha comentado anteriormente, se puede añadir los parámetros al MySqlCommand sin tener que saber el nombre pero si sabiendo la posición que ocupan en el procedimiento. Para que esto sea realmente válido para cualquier número y tipo de parámetro, se deberá usar un array de tipo object usando la palabra clave params https://msdn.microsoft.com/es-es/library/w5zay9db.aspx.

```
static void InvocarProcedimiento(string nombreProc, params object[] valores)
{
    try
    {
        //abrir la conexion
        MySqlConnection conexion = new MySqlConnection();
        conexion.ConnectionString = "Server=127.0.0.1;Port=3306;DataBase=prueba;Uid=xusa;Pwd=1234;";
        conexion.Open();
        MySqlCommand cmd = new MySqlCommand();
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Connection = conexion;
        cmd.CommandText = nombreProc;

        //asignar los parámetros
        MySqlCommandBuilder.DeriveParameters(cmd);
        for (int i = 0; i < cmd.Parameters.Count; i++) cmd.Parameters[i].Value = valores[i];
        cmd.ExecuteNonQuery();
        conexion.Close();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Para poder usar este método se tendrá que acceder al procedimiento con el método DeriveParameters, esto nos volcará en el array Parameters los argumentos del procedimiento por orden de aparición. Después solamente tendremos que recorrer este array para asignarle el valor que queramos, como he dicho con anterioridad para que este método tenga sentido lo suyo es utilizar el array de objetos con el modificador params. Donde la llamada al método InvocarProcedimiento sería de la forma:

```
InvocarProcedimiento("insertar", 222, "Prueba2", 777);
```

Aunque también podríamos acceder a cada uno de ellos directamente:

```
cmd.Parameters[0].Value = 999;
cmd.Parameters[1].Value = "pppp";
cmd.Parameters[2].Value = 999;
```

1.7. Realizar la llamada a una función

Igual que para los procedimientos, la función invocada deberá estar añadida a la Base de Datos y además no podrá tener el mismo nombre que un procedimiento, porque aunque MySql te lo permite el código C# no hace distinción entre los dos tipos y se produce excepción.

El código de la función referido a este ejemplo será el siguiente:

```
DELIMITER $$
CREATE FUNCTION finsertar ( a INT, b VARCHAR(25), c INT) RETURNS VARCHAR(25)
begin
  insert into tabla1 values (a, b, c);
  RETURN 1;
end $$
DELIMITER ;
```

Como la única diferencia entre un método y una excepción es que el método devuelve un valor, solo tendremos que tener encuenta que el primer valor que devuelve el array de parametros es el referente al valor de retorno de la función y podremos referirnos a el de dos formas direntes:

1. Parameters[0].Value
2. Parameters[“@RETURN_VALUE”].Value

```
//setear parametros del command
cmd.CommandType = CommandType.StoredProcedure;
cmd.Connection = conexion;
cmd.CommandText = nombreFunc;
MySqlCommandBuilder.DeriveParameters(cmd);
for (int i = 1; i < cmd.Parameters.Count; i++) cmd.Parameters[i].Value = valores[i - 1];
//ejecutar el query
cmd.ExecuteNonQuery();
//ver valor de retorno de dos formas diferentes
Console.WriteLine(cmd.Parameters["@RETURN_VALUE"].Value.ToString());
Console.WriteLine(cmd.Parameters[0].Value.ToString());
conexion.Close();
```