## 2.12 Enlace componentes a origen datos.

ADO.NET es la API de acceso a fuentes de datos de .NET.

Las características principales de ADO.NET son:

a) Permite trabajar tanto de forma conectada como desconectada del origen de datos. Una de las ventajas del acceso desconectado es que se consigue una mayor escalabilidad debido a que las conexiones de los usuarios no se mantienen

por periodos largos, con lo que se puede permitir el acceso de más usuarios.

b) Tiene una fuerte integración con XML, lo que lo hace más fácil de compartir, interpretar y trabajar con él.

c) Es independiente del lenguaje de programación utilizado.

d) No sólo permite acceder a bases de datos, sino también a otras fuentes de datos como hojas de cálculo, XML, texto, etc.

ADO.NET permite trabajar de dos formas:

• Acceso conectado: requiere una conexión permanente con la base de datos.

• Acceso desconectado: un subconjunto de los datos de la fuente de datos se copia en un dataset y, si luego se producen cambios en el dataset, éstos se propagan a la fuente de datos.

Diagrama

Descripción generada automáticamente

Las clases principales de ADO.NET son las siguientes:

• ***Connection***: realiza la conexión con una fuente de datos.

• ***Command***: permite hacer consultas u operaciones de modificación contra una fuente de datos.

• ***DataReader***: permite acceder a los resultados de una consulta realizada contra una fuente de datos.

• ***DataAdapter***: lee los datos y los carga en el *dataset*. Si ahí se producen cambios, el data adapter será el encargado de sincronizar todos los cambios en la fuente de datos.

• ***DataSet***: permite almacenar y manipular datos en memoria. Para establecer una **conexión con la fuente de datos**, en primer lugar, hemos de seleccionar el proveedor de datos adecuado, y a continuación tener en cuenta las cadenas de conexión.

**Proveedor de Datos**

Un proveedor de datos consta de un conjunto de clases que permiten el acceso y la comunicación con las fuentes de datos. Dentro del .NET Framework, se incluyen muchos proveedore de datos que no vienen integrados con el .NET Framework, tanto gratuitos como comerciales, entre ellos:

• **Proveedor de datos para ODBC**: proveedor genérico que permite acceder a cualquier fuente de datos mediante el driver ODBC correspondiente (necesario tenerlo instalado previamente).

• **Proveedor de datos para OLEDB**: proveedor genérico que permite acceder a cualquier fuente de datos mediante el driver OLE DB (necesario tenerlo instalado previamente). Es más rápido que ODBC.

• **Proveedor de datos para SQL Server**: proveedor específico para SQL Server. La comunicación con SQL Server tiene lugar sin capas intermedias (ni OLEDB ni ODBC), con lo que el rendimiento es aún mejor.

• **Proveedor de datos para Oracle**: es un proveedor de datos específico para Oracle, que también ofrece el mayor rendimiento posible, al no hacer uso de ninguna capa intermedia (ni OLE DB ni ODBC).

Todos los proveedores de datos han de ofrecer las siguientes clases:

Tabla

Descripción generada automáticamente

**Cadenas de conexión**

La cadena de conexión es una cadena de caracteres que identifica las características de la conexión con una cierta fuente de datos. La cadena de conexión incluye la localización del servidor, el nombre de la base de datos, el usuario, el *password*, u otras opciones específicas del proveedor de datos utilizado. La

siguiente tabla muestra los parámetros más comunes:

Tabla

Descripción generada automáticamente

Por ejemplo, el siguiente código se conecta a una base de datos SQL Server con el usuario sa, *password* xxx, y accede a la base de datos Northwind del servidor miBBDD:

SqlConnection myConnection = new SqlConnection();

myConnection.ConnectionString = "Data Source=miBBDD;" +

"Initial Catalog=Northwind;" +

"User ID=sa;Password=xxx"

myConnection.Open();

**Acceso conectado**

El acceso conectado consiste en abrir una conexión con la fuente de datos, ejecutar una serie de sentencias SQL y cerrar la conexión. La clase Command ofrece los siguientes métodos para ejecutar las sentencias:

Tabla

Descripción generada automáticamente

**Sentencias que no devuelven valores**

Utilizaremos el método *ExecuteNonQuery* para ejecutar sentencias SQL que no devuelven valores. El siguiente ejemplo crea una tabla llamada A:

SqlCommand cmd = new SqlCommand (

"CREATE TABLE A (A INT, PRIMARY KEY (A))", conn);

cmd.ExecuteNonQuery ();

En el caso de las sentencias SQL de manipulación (DML), el método Execute-NonQuery devuelve el número de filas afectadas. El siguiente ejemplo modifica el precio de los productos y obtiene el número de filas afectadas:

SqlCommand cmd = new SqlCommand (

"UPDATE PRODUCTOS SET precio = precio + 10", conn);

int numFilas = cmd.ExecuteNonQuery ();

**Sentencias que devuelven un único valor**

El método *ExecuteScalar* lo utilizaremos cuando la sentencia devuelva un único valor. El siguiente ejemplo devuelve el número de filas de una tabla:

SqlCommand cmd = new SqlCommand (

"SELECT COUNT(\*) FROM PRODUCTOS", conn);

int numFilas = (int)(cmd.ExecuteScalar ());

**Sentencias que devuelven un conjunto de valores**

En general, la ejecución de una sentencia *SELECT* o un procedimiento almacenado devuelve un conjunto de datos. Para ejecutar este tipo de sentencias utilizamos el método *ExecuteReader* de la clase *Command*, que devuelve un objeto

*DataReader* que nos permite consultar las filas obtenidas.

Un *DataReader* es un cursor que permite iterar hacia delante sobre un conjunto de filas (no se pueden recuperar elementos anteriores), resultado de la ejecución de una sentencia SQL o procedimiento almacenado. Para poder utilizar

un *DataReader* es necesario mantener abierta la conexión con la base de datos; no podemos cerrarla y después recorrer el *DataReader*. Por eso, se denomina acceso a datos conectado.

En el siguiente ejemplo, realizamos una consulta y obtenemos los resultados:

SqlCommand cmd = new SqlCommand(

"SELECT id, descr FROM usuarios", conn);

SqlDataReader reader = cmd.ExecuteReader();

while (reader.Read())

{

int id = reader.GetInt32(0);

String descr = reader.GetString(1);

}

reader.Close();

conn.Close();

La selección de datos es la operación que se realiza con mayor frecuencia sobre una base de datos. Esta es muy sencilla de implementar mediante los objetos SqlConnectiony SqlCommand. Tras la creación y la asociación de un objeto SqlCommand a una conexión, basta con asignar valor a la propiedad CommandText del comando con una consulta SQL para tener un comando de selección listo para ejecutarse.

Comando.CommandText = "SELECT \* FROM Alumnos";

La ejecución de consultas de selección de datos se realiza gracias al método ExecuteReader del objeto SqlCommand. Este método devuelve un objeto de tipo SqlDataReader gracias al cual es posible recuperar cada uno de los registros del resultado de la consulta, uno por uno, y únicamente avanzando en la colección de registros.

SqlDataReader reader = comando.ExecuteReader();

La lectura de un registro se realiza mediante la llamada al método Read del objeto SqlDataReader. Este método devuelve un valor booleano que indica si ha sido posible leer algún registro. Si es así, el objeto SqlDataReadercontiene los datos del registro, que pueden manipularse mediante el operador [].

reader.Read(); //Lectura del valor de la columna "Nombre" del registro string nombre = (string)reader["Nombre"];

Es necesario realizar una conversión de tipo de los valores recuperados de esta manera, pues están expuestos mediante el objeto SqlDataReader con forma de objeto object. Es posible, a su vez, recuperar el valor deseado fuertemente tipado mediante alguno de los métodos especializados de la clase SqlDataReader: el método GetString para obtener una cadena de caracteres, GetInt32 para obtener un valor entero, GetDateTime para obtener una fecha, etc. Estos métodos reciben como parámetro un valor entero que representa el índice de la columna en el registro.

reader.Read(); //Lectura del valor de la columna situada en la primera posición //del registro

string nombre = reader.GetString(0) while (reader.Read())

La lectura de la totalidad de los datos devueltos se realiza mediante un bucle while donde la condición de entrada es el valor devuelto por la llamada al método Read.

while (reader.Read()) {

Console.WriteLine("Nombre = {0}", reader.GetString(0)); }

Cuando la consulta no devuelve ningún valor, la lectura del resultado puede simplificarse utilizando el método ExecuteScalar del comando. Este devuelve un valor de tipo object que debe convertirse al tipo deseado.

SqlCommand comando = conexión.CreateCommand(); comando.CommandText = "SELECT COUNT(\*) FROM Alumnos"; int numAlumnos = (int)comando.ExecuteScalar(); Console.WriteLine("Hay {0} alumnos", numAlumnos);

La ejecución de una consulta para agregar, eliminar o actualizar datos se realiza de forma similar a la selección de datos. Se crea un comando que contiene una consulta UPDATE, INSERT o DELETE y se asocia a una conexión.

SqlCommand comando = conexión.CreateCommand();

comando.CommandText = "INSERT INTO Alumnos (Nombre, Apellidos) VALUES

(’Antonio’, ’AMARO SANZ’)";

Una vez definido el comando, la llamada al método ExecuteNonQuery del objeto SqlCommand envía la consulta al servidor de bases de datos. El valor devuelto es un valor entero que representa el número de registros afectados por la consulta.

int numeroRegistrosAfectados = comando.ExecuteNonQuery();

Los comandos que modifican elementos de la estructura de la base de datos (ALTER TABLE, por ejemplo) se ejecutan, también, mediante este método.

**Comprobación de valores *null***

Podemos comprobar cuándo el valor de una columna es nulo (es decir, es un valor *null*) con el método *IsDbNull* del objeto *DataReader*, como podemos ver en el siguiente ejemplo:

if (!reader.IsDBNull (3)) // Si la tercera columna no es null

{ int i = dr.GetInt32(3);

}

Gracias a los tipos anulables (pueden contener valor nulo y se indican añadiendo ? al tipo), podemos ejecutar el método *GetXXX* sin comprobaciones adicionales:

int? i = dr.GetInt32(3); // Si la tercera columna no es null

(más información del método GetInt

<https://docs.microsoft.com/es-es/dotnet/api/system.data.datatablereader.getint32?view=net-6.0>)

**Acceso desconectado**

**Datasets no tipados**

Un *DataSet* es una copia en memoria de una parte de una base de datos. Constituye una vista “desconectada” de los datos, es decir, existe en memoria sin una conexión activa a una base de datos que contenga la correspondiente tabla o vista. El objeto *DataSet* guarda los datos y su estructura en XML, lo que permite por ejemplo enviar o recibir un XML por medio de HTTP.

Un *DataSet* está compuesto por tablas (*DataTables*) y una lista de relaciones entre éstas (*Relations*). Además, el *DataSet* mantiene un *schema* XML con la estructura de los datos. Un *DataTable*, a su vez, está formado por columnas (*DataColumn*) y contiene una serie de filas de datos (*DataRow*). También puede tener definidas una serie de restricciones (*Constraints*), como *Primary key*, *Foreign key*, *Unique* o *Not null*.

Las filas constituyen los datos que se almacenan en el *DataTable*. De cada *DataRow* se guardan dos copias, una de la versión inicial (la que se recupera de la base de datos), y otra de la versión actual, para identificar los cambios realizados en el *DataSet*.

Diagrama

Descripción generada automáticamente

El siguiente ejemplo crea un objeto DataTable que representa la tabla PRODUCTOS de una fuente de datos, y lo añade al DataSet ALMACEN:

DataSet ds = new DataSet("ALMACEN");

DataTable dt = ds.Tables.Add("PRODUCTOS");

dt.Columns.Add("id", typeof (Int32));

dt.Columns.Add("nombre", typeof (String));

dt.Columns.Add("precio", typeof (Double));

Para recuperar tablas o columnas de las colecciones Tables y Columns, podemos

acceder por posición o por el nombre del elemento. En el siguiente ejemplo,

recuperamos la columna “id” de la tabla PRODUCTOS:

// acceso por posición

DataColumn dc = ds.Tables[0].Columns[0];

// acceso por nombre

DataColumn dc = ds.Tables["PRODUCTOS"].Columns["id"];

La clase DataTable provee distintas maneras de acceder a los datos que contiene. La más sencilla consiste en utilizar su colección Rows. Es posible tratar cada uno de los elementos almacenados en esta colección utilizando un bloque foreach.

foreach (DataRow row in tabla.Rows) {

Console.WriteLine("El identificador del registro es {0}", row.ItemArray[0]); }

La lectura de cada elemento del registro a partir de un índice puede resultar una operación algo delicada, en particular si se modifica el orden de las columnas de la tabla. El uso del método de acceso basado en el nombre de la columna es, naturalmente, mucho mejor.

foreach (DataRow row in tabla.Rows) {

Console.WriteLine("El identificador del registro es {0}", row["Identificador"]); }

Un último medio para leer los datos de un objeto DataTable consiste en crear un DataTableReader, que es un tipo derivado de DbDataReader y que provee varios métodos especializados para la lectura de datos con la gestión de un tipado fuerte. La clase DataTableReader se utiliza de la misma manera que el tipo SqlDataReader:

DataTableReader dataTableReader = tabla.CreateDataReader();

while (dataTableReader.Read()) {

Console.WriteLine("El identificador del registro es {0}", dataTableReader[0]); }

**Ejercicio: Instalar SQL Server**