6

Acceso a Bases de Datos

El acceso a almacenes de información, ya sean éstos bases de datos típicas o de cualquier otra clase, es una necesidad que tienen la mayoría de proyectos.

Introducción

Muchas páginas web necesitan almacenar y recuperar los datos con los que trabajan habitualmente, usando para ellos distintos recursos. Uno de ellos, el más corriente, consiste en acceder a un **RDBMS** (Relational Data Base Management System), un software específico para la gestión de datos. Existen distintos productos de diferentes fabricantes en esta categoría, como **Oracle**, Microsoft **SQL Server**, Microsoft **Access**, **IBM DB2** y similares.
Otras opciones son sistemas de almacenamiento local que, básicamente, se comportan como si fuesen un **RDBMS** simple. Un ejemplo es Access. También puede alojarse la información en hojas de cálculo Excel, archivos en formato **XML** (Extensible Markup Language) o, incluso, archivos de texto con los datos separados por comas, tabuladores o algún otro carácter.
La plataforma Microsoft .NET cuenta con un conjunto de servicios, denominados genéricamente como **ADO.NET**, que facilitan el acceso a distintas fuentes de datos. A su vez, el entorno de Visual Web Developer .dispone de asistentes y diseñadores específicos para facilitar esta tarea.

El modelo de objetos de ADO.net

ADO.NET se compone de unas decenas de clases e interfaces distribuidas por varios espacios de Nombres (Namespaces). Algunas de ellas son genéricas, de tal forma que sólo habrá que aprender a usarlas una vez. Otras, como las de los proveedores específicos, se aplican a cada tipo de origen de datos. VWD cuenta con varios proveedores de datos, uno para trabajar con SQL Server, otro para OLE DB, otro para Oracle y otro para ODBC.

En el Cuadro de herramientas, concretamente en la página Datos, encontramos tres componentes para acceder a un origen de datos, sqlDataSource, para acceder a una BD de Access, AccessDataSource o para acceder a un fichero XML, XMLDataSource.

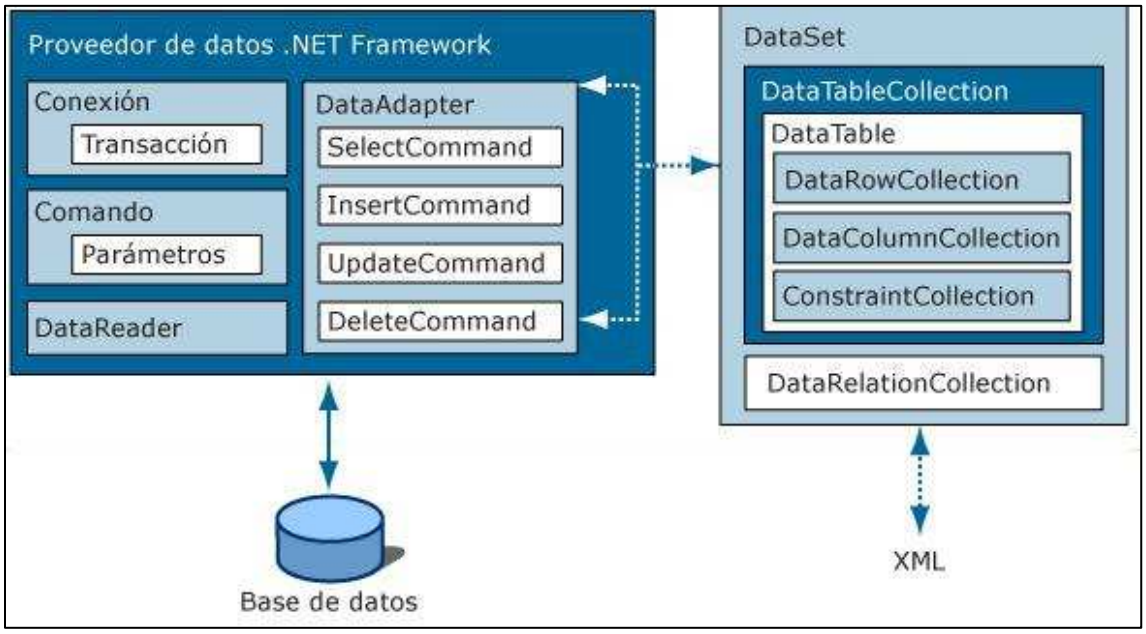
Ámbitos relacionados con ADO.NET

Las interfaces y clases genéricas de ADO.NET se alojan en los ámbitos **System**.**Data** y **System**.**Data**.**Common**. Algunas de esas interfaces se implementan en clases específicas de cada proveedor, mientras que otras cuentan con una implementación genérica.
Cada proveedor cuenta con uno o más ámbitos, como son **System**.**Data**.**SqlClient**, **System**.**Data**.**OracleClient**, **System**.**Data**.**OleDb** y **System**.**Data**.**Odbc** que, como puede suponer, alojan las clases de los proveedores para SQL Server, Oracle, OLE DB y ODBC. Además tenemos también el ámbito **System**.**Data**.**SqlTypes**, en el que se definen tipos a medida de SQL Server.

Esquema de funcionamiento.

Independientemente del proveedor concreto que vayamos a usar, el esquema de funcionamiento de ADO.NET es siempre homogéneo y se basa en el uso de las mismas categorías de componentes, ocupándose la implementación específica de los detalles de bajo nivel.

El primer paso será siempre el establecimiento de una conexión con la fuente de datos, utilizando para ello un elemento denominado **Connection.** Será el momento de establecer atributos que, por ejemplo, identifiquen al servidor, nombre de usuario para iniciar la sesión, y en algunas Bases de Datos como SQL Server, el nombre de la base de datos.
También se pueden definir otros parámetros avanzados de la conexión como si es de sola lectura, timeout de la misma...
Acto seguido pasaremos a crear un comando definiendo la acción a ejecutar sobre esa conexión. Esto lo haremos con un objeto de la clase **Command.** Básicamente, pueden seguirse dos caminos diferentes: ejecutar un comando que devuelva un lector de datos, representado por la interfaz **IDataReader**, o bien usar un adaptador de datos para generar un conjunto de datos o **DataSet**.
Los lectores de datos se utilizan cuando sólo se precisa la lectura, no modificación, de los datos, y siempre de manera secuencial. Es el método ideal si, por ejemplo, van a recuperarse datos para generar un informe o elaborar unos resultados.
Si tenemos necesidad de editar la información, mostrarla al usuario en formularios, ya sean Windows o Web, y efectuar otras tareas habituales de manipulación, usaremos generalmente conjuntos de datos. Cada proveedor cuenta con un adaptador capaz de generar un conjunto de datos, un objeto que, a partir de ese momento, no guarda ya enlace alguno con la fuente original de información.



1º PASO: Configuración de conexiones

Centrémonos ahora en el primer paso: la conexión con la fuente o depósito donde se encuentra almacenada la información. Con este fin utilizaremos la clase **SqlConnection**, **OracleConnection**, **OleDbConnection** u **OdbcConnection** según el proveedor que vayamos a usar. En nuestro caso lo más utilizado será **SQLConnection** y **OledbConnection**. Intentaremos hacer alguna prueba con **OracleConnection**. Aunque la forma de trabajar es exactamente igual en todas ellas. Todas estas clases implementan la interfaz **IDbConnection** en la que se establece la existencia de la propiedad **ConnectionString** y los métodos **Open**() y **Close**(), entre otros miembros.
Mediante la cadena de conexión se identifica la localización del origen de los datos, teniendo un formato u otro según el proveedor que utilicemos. Para conectar con una base de datos **SQL Server** mediante un componente **SqlConnection**, por ejemplo, indicaríamos el nombre de la maquina donde se encuentra el servidor de datos, el método de autenticación en el mismo (Usuario y contraseña si hicieran falta) y el nombre de la base de datos. Para trabajar con **Access 2003** mediante OLE DB, por el contrario, la cadena de conexión debería indicar el proveedor, en este caso sería Microsoft.Jet.OLEDB.4.0, y el camino y nombre completo del archivo .**mdb** donde están los datos.

Establecida la cadena de conexión, la llamada al método **Open**() intentaría conectar con el servidor o abrir el archivo, según los casos.
Análogamente, el método **Close**() cerraría esa conexión.

Cadenas de conexión.

Una información fundamental para poder acceder mediante ADO.NET a un depósito de datos, sea cual sea su tipo, es la cadena de conexión. La mejor manera de comprender su estructura y composición es ver algunos ejemplos.
Usando el proveedor de SQL Server, representado por el componente **SQLConnection**, la cadena debe indicar el nombre del servidor de datos, el nombre de la base de datos y facilitar la información de inicio de sesión: usuario y clave, a menos que se utilice la seguridad integrada de Windows. Un ejemplo podría ser el siguiente:

```
SqlConnection conn = new SqlConnection();
conn.ConnectionString = "Data Source=.\\SQLEXPRESS;AttachDbFilename='C:\\SQL
Server 2000 Sample Databases\\PUBS.MDF';Integrated Security=True;Connect
Timeout=30;User Instance=True";
conn.Open();
```

En este caso, se ha asignado la cadena de conexión a la propiedad ConnectionString del objeto conn. Tambien podría haberse asignado directamente al constructor:

```
SqlConnection conn = new SqlConnection("Data
Source=.\\SQLEXPRESS;AttachDbFilename='C:\\SQL Server 2000 Sample
Databases\\PUBS.MDF';Integrated Security=True;Connect Timeout=30;User
Instance=True");
conn.Open();
```

La **cadena de conexión** indica que existe un ordenador, llamado **SQLEXPRESS**, al cual tenemos acceso mediante nuestra infraestructura de red y que ejecuta el servidor de datos SQL Server. Dicho servidor contiene una base de datos denominada **PUBS.MDF** en la cual iniciaremos sesión utilizando la seguridad integrada de Windows. Cuando hablamos de seguridad integrada de Windows, queremos decir que la identificación del usuario se delega al sistema operativo Windows. Y dicha identidad es la que tratará SQL Server como usuario de conexión.
Para abrir una base de datos Access 97 utilizando OLE DB, y asumiendo que el archivo se llama BddDatos.mdb, la cadena de conexión sería:

Proveedor OleDB para acceso a Bases de datos de MS Access 97

```
OleDbConnection conn = new OledbConnection();
conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;
Data Source=BddDatos.mdb”;
conn.Open();
```

Proveedor OleDB para acceso a Bases de datos de Oracle

```
OleDbConnection conn = new OledbConnection();
conn.ConnectionString=”Provider=MSDAORA; Data Source=zabal; User ID=martat;
Password=zabal”;
conn.Open();
```

Proveedor OleDB para acceso a Bases de datos de MS Access 2000 o superior

```
OleDbConnection conn = new OleDbConnection();

conn.ConnectionString=”Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=c:\bin\LocalAccess40.mdb;”
conn.Open();
```

Proveedor OleDB para acceso a Bases de datos de SQL Server

```
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString=”Provider=SQLOLEDB;Data Source=MySQLServer;Integrated
Security=SSPI;”
conn.Open();
```

El componente **OleDbConnection** necesita saber qué proveedor **OLEDB** va a utilizarse, indicando el nombre completo con la clave **Provider**. En este caso seleccionamos el proveedor Jet, motor de Access, pero que también puede utilizarse para, por ejemplo, abrir una hoja de cálculo Excel o un archivo de texto.

Es conveniente cerrar una conexión de base de datos una vez utilizada. Para ello se debe emplear el método **Close()** del objeto conexión.

Acceder a los datos de una consulta

Para acceder a los datos de una consulta, los pasos a dar son los siguientes:

- Crear y abrir una conexión con la base de datos
- Crear un comando de base de datos que represente a la sentencia SQL que quiere ejecutar
- Ejecutar el comando con el método **ExecuteReader()** que retorna un objeto **DataReader**
- Acceder a los datos del objeto **DataReader**

Un objeto **DataReader** representa un cursor que sólo permite recorrer los datos hacia delante.
Para leer cada fila del objeto se emplea el método **Read()**. Este método devuelve un valor **False** cuando se alcanza el final de los datos, por tanto para saber si la instrucción SQL ha devuelto valores o no puede meterse este método en una condicional:

if (dtrrautores.Read()){
……..
……..
}

Si la instrucción de SQL ha devuelto más de una fila podemos emplear dicha instrucción en un bucle repetitivo:

while (dtrautores.Read()){
…….
……..

}

Para acceder a los datos obtenidos por el método **Read** podemos emplear varios mecanismos distintos. Suponiendo que el objeto **SqlDataReader** se llama **dtrautores**:

- **dtrautores("au_id")**: Retorna el contenido del campo especificado como una cadena.
- **dtrautores.Item("*au_id*")**: Similar al anterior.
  dtrautores(0): Retorna el contenido del campo cuya posición se indica como una cadena.
- **dtrautores.Get*TipoCampo*(0)**: Retorna el contenido del campo especificado como un dato del tipo indicado. Hay un método por cada tipo de datos: GetBoolean, GetByte, GetChar, GetdateTime, GetDouble, GetFloat, etc.

Ejemplo
En este ejemplo se van a recuperar los datos de la tabla **authors** de la base de datos **pubs** y se van a mostrar en una etiqueta.

El código:

```
using System.Data.SQLClient

       Protected void Page_Load (Object sender, e SystemEventArgs){
              SqlConnection conPubs;
              SqlCommand cmdAutores;
              SqlDataReader dtrAutores;
              conPubs= new SqlConnection();
              conn.ConnectionString = "Data
              Source=.\\SQLEXPRESS;AttachDbFilename='C:\\SQL Server 2000 Sample
              Databases\\PUBS.MDF';Integrated Security=True;Connect
              Timeout=30;User Instance=True";
              conn.Open();
              cmdAutores = new SqlCommand( _
```

```
                  "Select au_lname, au_fname From Authors", conPubs);
            dtrAutores = cmdAutores.ExecuteReader()
            lblListado.Text = "" ;
            While (dtrAutores.Read()){
            lblListado.Text += "<li>" & dtrAutores("au_lname") +
                                          ", " + dtrAutores("au_fname");
            }
            dtrAutores.Close();
            conPubs.Close();
      }

<html><head><title>Ejemplo </title></head>
<body>
<h2>Listado de Autores</h2>
<asp:Label
      ID="lblListado"
      RunAt="Server" />
</body>
</html>
```

MODO CONECTADO

Definición de comandos

Las clases **Command** cuentan con diversos constructores que facilitan, en el momento de crear el objeto, el establecimiento de múltiples propiedades con las que se enlaza el comando con la conexión, define el tipo o el texto del comando. También es posible crear un comando, directamente asociado a una conexión, utilizando el método **CreateCommand**() con el que cuentan las clases **Connection**.
El texto del comando, asignado a la propiedad **CommandText**, suele ser una consulta en lenguaje SQL, si bien caben otras posibilidades como la recuperación de todo el contenido de una tabla simplemente facilitando su nombre o la ejecución de un procedimiento almacenado.
Será la propiedad **CommandType** la que indique cómo interpretar **CommandText**.
Preparando el comando, se utilizará alguno de los métodos **Execute**() para ejecutarlo y obtener un resultado. Como en este caso lo que nos interesa es obtener un lector, el método a utilizar sería **ExecuteReader**(). Este método instanciará un objeto de tipo **DataReader** y además lo rellenará con los datos que cumplan las condiciones del objeto comando.

Columnas, filas y valores.

Una vez contemos con el objeto **DataReader**, podemos consultar varias propiedades para saber cuántas columnas hay disponibles, cuáles son sus nombres o recuperar los valores que contienen en la fila actual.
La propiedad **FieldCount** nos indica el número de columnas, dato que puede utilizarse como límite de los índices de los métodos **GetName()** y **GetValue()**. El primero facilita el nombre o título de una columna, mientras que el segundo entrega el valor que contiene en la fila actual. En parámetro de estos métodos será un número comprendido entre 0 y el valor de **FieldCount menos 1**. El método **GetOrdinal()** nos devuelve el número de la columna si le pasamos el nombre de la misma como parámetro.
Al ejecutar el comando y obtener el lector, aún no se ha recuperado ninguna fila de datos del origen. Es necesario llamar al método **Read()** para recuperar una fila, repitiendo ese paso hasta que el valor devuelto por el mismo **Read()** indique que ya no hay más datos.

```
cn.Open();
OleDbCommand cmd = new OleDbCommand();
OleDbDataReader   rdr;
cmd.CommandText = "SELECT CategoryName FROM CATEGORIES";
cmd.Connection = cn;
rdr = cmd.ExecuteReader();
while (rdr.Read()){
        lst.Items.Add(rdr.GetValue(rdr.GetOrdinal("Category_id")))
}
cn.Close();
```

Comandos que devuelven un valor

Utilizando el método **ExecuteScalar** podemos ejecutar sentencias SQL tipo SELECT que en lugar de devolver un conjunto de registro devuelven un único valor. Funciones Count, Max, Min, Sum, Avg,...

```
cn.Open();
OleDbCommand cmd =new OleDbCommand();
cmd.CommandText = "SELECT count(*) FROM CATEGORIES";
cmd.Connection = cn;
txt1.text=cmd.ExecuteScalar();
cn.Close();
```

Uso de comandos para actualizar la BD

A la hora de modificar los datos de la Base de Datos, uno de los mecanismos más utilizados es el uso de comandos. Por ej. El método **ExecuteNonQuery** de la clase **OleDBCommand** nos permite ejecutar sentencias SQL (**DELETE**, **INSERT**, **UPDATE**) directamente sobre la BD.
Este método nos devolverá el número de filas afectadas por el comando.

```
cn.Open();
OleDbCommand cmd=new OleDbCommand();
cmd.CommandText = "INSERT INTO CATEGORIES " & _
"(CATEGORYNAME) VALUES ('Categoria')";
cmd.Connection = cn;
cmd.ExecuteNonQuery();
cn.Close();
```

Además nos permiten parametrizar los comandos, de tal forma que podemos utilizar la misma sentencia SQL con diferentes valores. Valores que habitualmente introduce el usuario. Para realizar esta parametrización debemos identificar con una **?** todos los valores que deseamos parametrizar. A continuación daremos un nombre y un valor a los diferentes parámetros. Es importante que el tipo del valor del parámetro coincida con el tipo del campo de la Base de datos. Y para finalizar ejecutaremos el comando.

```
cn.Open();
OleDbCommand cmd=new OleDbCommand();
cmd.CommandText = "DELETE FROM CATEGORIES WHERE " & _
"CATEGORYNAME=?";
cmd.Connection = cn;
cmd.Parameters.Add("pCategoryName", txtcategoria.text);
cmd.ExecuteNonQuery();
cn.Close();
```

Uso de parámetros en el objeto Command

Muchas veces el comando a ejecutar en la Base de Datos no es siempre el mismo, sino que depende de ciertos valores introducidos por el usuario. Esto nos obliga a parametrizar dicho objeto comando. Para conseguir esta parametrización tenemos que definir una sentencia SQL indicando claramente cuales son los valores a parametrizar y posteriormente añadir los parámetros a la colección Parameters del objeto Comando.

Por ej. En una **Base de Datos SQL Server**:

```
SqlCommand cmd= new SqlCommand("select * FROM Authors where au_id=@autor", cn);
```

Identificamos que el campo au_id debe de tener un valor en concreto que el programador no conoce en el momento que está programando. En SQL Server los nombres de los parámetros se identifican colocándoles el carácter @ por delante.

```
cmd.Parameters.Add(New SqlParameter("@autor",txtau_id.text));
SqlDataReader rdr  = cmd.ExecuteReader();
```

O

```
SqlParameter paramautor = new SqlParameter("@autor",System.Data.SqlDbType.VarChar, 5);
paramautor.Value = txtautor.Text;
cmd.Parameters.Add(paramautor);
SqlDataReader rdr  = cmd.ExecuteReader();
```

Posteriormente añadimos el parámetro al objeto comando, identificando su nombre y su valor. Es **muy importante** que el tipo de dato del valor coincida con el tipo de dato del campo en la base de datos.

Por ej. **En una Base de datos Access**:

```
OleDBCommand cmd= new OleDBCommand ("select * FROM Authors where au_id=? ", cn);
```

Identificamos los parámetros con el símbolo de la interrogación. Es decir no se les da un nombre sino que a todos se les pone el símbolo de la interrogación.

```
cmd.Parameters.Add (new OleDBParameter("autor",txtau_id.text));
OleDBDataReader rdr = cmd.ExecuteReader();
```

O

```
OleDbParameter paramautor = new OleDbParameter("autor",System.Data.OleDbType.VarChar, 5);
paramautor.Value = txtautor.Text;
cmd.Parameters.Add(paramautor);
SqlDataReader rdr  = cmd.ExecuteReader();
```

Y al añadir el parámetro a la colección de parámetros se le da un nombre y un valor. Hay que tener en cuenta que si tenemos más de un parámetro, como al crear el comando no se le da un nombre al parámetro, al añadir el parámetro a la colección, lo identificará por posición. Es decir el primer parámetro añadido se identificará con la primera interrogación, el segundo con la segunda interrogación y así sucesivamente.

MODO DESCONECTADO

Conjuntos de datos (Dataset)

Los lectores de datos son una opción sencilla a la hora de recuperar información, pero no resultan útiles si nuestras necesidades van más allá. No es posible **acceder más que a filas individuales**, **siempre hacia delante**, y **no es factible la edición** puesto que no se devuelven cambios al origen de la información.
Los conjuntos de datos, nombre con el que se denominan los objetos de la clase **DataSet**, nos ofrecen todas esas posibilidades, además de una manera independiente del tipo de base de datos u origen con el que estemos trabajando. Para ello se apoya en varias clases, como los **adaptadores de datos**, los **generadores de comandos** y **clases** que representan **tablas** individuales y las **relaciones** existentes entre ellas.
Es posible generar conjuntos de datos a partir de información ya existente, por ejemplo en una base de datos, **así como crearlos partiendo desde cero y mediante código**. Los **DataSet** son tan flexibles que pueden utilizarse incluso como solución de almacenamiento de datos local, sin necesidad de contar con ningún proveedor.

Adaptadores de datos (DataAdapter).

Para generar un conjunto de datos a partir de una consulta, y poder posteriormente actualizar esos datos en el origen, necesitaremos un adaptador de datos, un objeto de una **clase DataAdapter**. Hay una clase de este tipo por cada proveedor y todas ellas implementan la **interfaz IDataAdapter**.
Un adaptador de datos se enlaza con un objeto **Connection**, del que toma los datos de conexión, y un objeto **Command**, del que toma el comando de selección de datos. Lo interesante es que dichos objetos pueden ser creados automáticamente por el propio adaptador, simplemente facilitando todos los datos necesarios. Por ejemplo:

```
OleDbDataAdapter  Adaptador = new OleDbDataAdapter  (“SELECT * FROM Categorías”,conn);
```

siendo conn un objeto del tipo **OledbConnection** con una cadena de conexión creada a una Base de Datos y habiendo abierto la conexión anteriormente.

Para **generar el conjunto de datos** no hay más que llamar al método **Fill()** del adaptador, facilitando como parámetro un objeto **DataSet** creado previamente. Esta acción se traducirá en la apertura de la conexión, ejecución del comando, recuperación de los datos y generación en el **DataSet** de los **DataTable** que correspondan, representando las tablas de datos.

```
ds = new DataSet;
adaptador.Fill(ds, "Categorias");
```

DataTable

Cada **DataTable** contiene una colección de filas (**Rows**) de la clase **DataRow** y una colección de columnas (**Cols**) de la clase **DataColumn**. A través de un **DataRow** accederemos a los valores de las diferentes filas del **DataTable** y a través de un **DataColumn** a las definiciones de las diferentes columnas del **DataTable**.

Por ej. Para recorrernos todas las filas de una **DataTable** podríamos utilizar este código:

```
DataRow dr ;
For Each (dr in ds.Tables[0].Rows){
    txt1.text += (dr["NombreCategoría"].ToString()) + “-“;
}
```

Y para recorrernos todos los nombres de los campos de un **dataTable** podemos utilizar este otro:

```
DataColumn dc;
Foreach (dc In ds.Tables[0].Columns){
    txt1.text +=  (dc.ColumnName) + "-";
}
```

Para mostrar el primer registro de una tabla lo haremos de la siguiente manera:

```
DataTable dt=ds.Tables[0] ;   ' o   DatatTable dt =ds.Tables["Categorias"]
txtIdcategoria.Text = dt.Rows[0]["IdCategoría"].ToString();
txtnombrecategoria.Text = dt.Rows[0]["Nombrecategoría"].ToString();
txtdescripcion.Text = dt.Rows[0]["Descripción"].ToString();
```

Para poder movernos por todos los registros de la **DataTable** solo tendremos que incrementar o decrementar el número de fila a mostrar. Sabemos que en VB .NET el primer registro siempre es el número **0** y el último de la colección de filas es **dt.Rows.Count-1**.

Seleccionar varios registros de un DataTable (Método Select)

Además vamos a disponer de un método para poder filtrar y ordenar nuestros registros: el **método Select.**

El método **Select** del objeto **DataTable** devuelve una colección de objetos **DataRow** con los registros seleccionados.
La **sintaxis** de este **método** es:

objetoDataTable.Select(*expresiónFiltro*)

Esta instrucción **obtendrá** un **conjunto de filas** que cumplan una determinada condición.

objetoDataTable.**Select**(*expresiónFiltro,expresiónSort)*

Esta instrucción obtendrá un **conjunto de filas** que cumplan una condición ordenados a partir de uno o varios campos.

ExpresiónFiltro es una cadena con una expresión lógica que contiene el criterio de búsqueda en el siguiente formato: ***nombreColumna opRelación valor***

La columna y el valor deben pertenecer al mismo tipo de dato.
Los operadores de relación serán =, <,>,<=,>=,<> o LIKE.
Es posible unir varias expresiones con los operadores AND u OR.

Por ejemplo:

Para valores numéricos:	Para valores de cadena, el valor se debe encerrar entre comillas simples.	Para valores de fecha, es necesario encerrar las fechas entre almohadillas.
"Precio > 100"	"Ciudad = 'Madrid'"	"Fecha > #23/04/2004#"

El operador LIKE compara una columna con un patrón. En el patrón se peden utilizar comodines como el % o el *.Por ejemplo: "Ciudad LIKE 'M%'"

expresiónSort es una cadena que contiene los nombres de las columnas por los que queremos ordenar separadas por comas.
El criterio de ordenación por defecto es ascendente. Cada nombre de columna puede ir seguido de las palabras reservadas ASC o DESC para indicar el tipo de ordenación (ascendente y descendente respectivamente).

Por ejemplo:

"Apellido"	"Apellido,Nombre"	"Provincia DESC, IdCliente"
Ordena por la columna apellidos (ascendente)	Ordena por las columnas Apellido y Nombre	Ordena descendentemente por Provincia y a continuación, de forma ascendente por IdCliente

```
'Filtrar los Clientes cuya provincia empiece por V y ordenarlos por apellidos y después por Nombre
String cadena;
DataRow dr;
Foreach (dr In ds.Tables[“Clientes”].Select("Ciudad LIKE 'M%'", "Apellido,Nombre")){
        cadena= dr["numclie"].ToString() + “-“ + dr.Item("Nombre").ToString() + “-“ + dr["repclie"].ToString() +
        “-“ + dr["limitecredito"].ToString()
        LstClientes.Items.Add (cadena);
}

'Buscar con Select un Id.Cliente introducido en un cuadro de texto
DataRow[] filas= ds.Tables["Clientes"].Select("IdCliente=" + txtIdCliente.Text)
If (filas.Length == 0){
        MessageBox.Show("No existe",”AVISO”)
}else{

        txtNomcliente.Text=filas[0]["numclie"].ToString();
        txtNombre.Text= filas[0]["nombre"].ToString();
        txtresponsable.Text=filas[0] ["repclie"].ToString();
        txtlimitecredito.Text=filas[0]["limitecredito"].ToString();
}
```

Buscar un solo registro de un DataTable (Método Find)

El método **Find** de la clase **DataRowsCollection** (la colección Rows es un dato de esa clase), permite localizar un único registro a partir de una clave principal.

Al realizar una búsqueda por clave únicas es más eficiente que el método **Select**, pero se necesita definir una clave primaria, lo que hace que este método se complique un poco más que el método **Select**. La clave primaria se creará con la propiedad **PrimaryKey** de la clase **DataTable**. Recibe como valor un **array de objetos DataColumn** con cada una de las partes de la clave.

El método **Find** devuelve un único objeto de tipo **DataRow** si localiza un registro que cumpla la condición o un valor nulo si no se encuentra.

Las columnas que se utilizan como clave no deben tener valores repetidos.

```
‘Para establecer el campo numclie como clave primaria.
DataColumn[] clave=new DataColumn[1];
Clave[0] = ds.Tables["Clientes"].Columns["numclie"];
ds.Tables["Clientes"].PrimaryKey = clave

‘Para establecer los campos Apellido y Nombre como clave primaria.
DataColumn[] claveApeNom=new DataColumn[2];
claveApeNom[0] = ds.Tables["Clientes"].Columns["Apellido"];
claveApeNom[1] = ds.Tables["Clientes"].Columns["Nombre"];
ds.Tables["Clientes"].PrimaryKey = claveApeNom
```

Ejemplo: Buscar en la tabla Pedidos por clie y FechaPedido(se supone que no hay valores repetidos).

```
'Establecer la clave primaria clie+Fecha

DataColumn[]claveIdFecha =new DataColumn[2];
claveIdFecha[0] = ds.Tables["Pedidos"].Columns["clie"];
claveIdFecha[1] = ds.Tables["Pedidos"].Columns["FechaPedido"];
ds.Tables["Pedidos"].PrimaryKey = claveIdFecha;
```

```
'Leer el Id.Cliente y la Fecha

Object [] valores=new Object[2];
valores[0] = txtIdCliente.Text;
valores[1] = Convert.ToDate(txtfechaPedido.Text);

Dim filaEncontrada As DataRow = ds.Tables["Pedidos"].Rows.Find(valores);
if (filaEncontrada != null) {
        lblaviso.Text=”Registro mno encontrado”;
} else {
        txtIdPedido.text=filaEncontrada["IdPedido"];
        txtIdcliente.text=filaEncontrada["clie"];
        txtfechPedido.text=filaEncontrada["FechaPedido"];
        txtproducto.text=filaEncontrada[“Producto"];
        txtcantidad.text=filaEncontrada["Cant"];
        txtimporte.text=filaEncontrada["Importe"];
}
```

Insertar un registro en un DataTable (Método NewRow)

Para insertar un registro en una tabla se utilizará el método **NewRow()** de un **DataTable**. Después añadiremos los valores introducidos en algún control de nuestro formulario en los distintos campos de la tabla y por último se añadirá la nueva fila al conjunto de filas de la tabla.

```
DataRow dr = ds.Tables("Categorias").NewRow()
    dr["IdCategoría"] = txtIdcategoria.Text;
    dr["Nombrecategoría"] = txtnombrecategoria.Text;
    dr["descripción"] = txtdescripcion.Text;
    ds.Tables["categorias"].Rows.Add(dr);
```

Borrar un registro en un DataTable (Método Delete)

Deberemos localizar el registro a borrar y una vez localizado se le aplicará el método **Delete()** sobre ese registro (fila).

```
DataRow  dr;
  dr = dt.Rows.Find(txtbuscar.Text);
     if (dr!=null){
        lblaviso.Text=”Registro no encontrado”;
     } else {
           dr.Delete();
    }
```

Actualizar un registro en un DataTable

Debemos localizar el registro a modificar y una vez modificado cambiar el valor de uno o varios de sus campos con algún valor pasado en algún control del formulario.

```
DataRow dr;
  dr = dt.Rows.Find(txtbuscar.Text);
     if (dr!=null){
        lblaviso.Text=”Registro no encontrado”;
     } else {
           dr["descripción"] = txtdescripcion.Text;
     }
```