

Programación con Delphi 6 y Kylix

Francisco Charte Ojeda

© EDICIONES ANAYA MULTIMEDIA (GRUPO ANAYA, S.A.), 2001

13

MyBase

En el undécimo capítulo se describió el proceso general para conectar con una base de datos usando diferentes mecanismos: ADO, BDE, DBX. En todos estos casos se asume que existe un motor de bases de datos y, en las configuraciones cliente/servidor y de múltiples capas, también un servidor de datos o RDBMS. Sin embargo, no todas las aplicaciones que necesitan almacenar los datos con los que trabajan habitualmente precisan tantas posibilidades como las que ofrecen esas opciones.

En estos casos, *MyBase* puede resultar ser la mejor alternativa. Las aplicaciones que utilizan *MyBase* no precisan de ninguna herramienta de diseño externa para la definición de la base de datos.

Para trabajar con los datos no son precisos controladores, como ocurre con DBX o BDE, por lo que la instalación es mucho más simple, así como la distribución del proyecto. Los datos se almacenan localmente y el formato es, a elección del programador, binario o XML. Por último, hay que destacar que *MyBase* está presente tanto en Delphi 6 como en Kylix, siendo una de las opciones a tener en cuenta si precisamos llegar a Windows y Linux con un mismo proyecto y los requerimientos de acceso a datos de éste se limitan a los de una configuración cliente simple.

Nuestro objetivo, en este capítulo, es facilitarle toda la información necesaria para que pueda aprovechar *MyBase*, evaluar cuándo es un sustituto apropiado para Paradox, dBase y formatos similares y efectuar las operaciones más habituales.

Aunque la mayoría de las imágenes mostradas en el capítulo pertenecen a Delphi 6, las operaciones descritas son idénticas en Kylix.

Generalidades sobre *MyBase*

Antes de entrar en detalles concretos y ejemplos prácticos de cómo utilizar el único componente implicado en este mecanismo de acceso a datos, vamos a introducir algunos conceptos generales que nos permitan tener una visión global de *MyBase*.

Una aplicación *MyBase* en ningún momento conecta con una base de datos para recuperar ni actualizar información. Todos los datos se alojan en un archivo simple en disco y, durante la ejecución del programa, deben recuperarse completamente en memoria. Esto implica que el tamaño de las bases de datos *MyBase* está limitado físicamente por la propia memoria del sistema. En contraste, una base de datos alojada en un servidor no tiene más límite que el del tamaño de los discos donde se encuentra almacenada. Hablamos de algunos megabytes, en el caso de *MyBase*, contra muchos gigabytes o, incluso, terabytes, en el caso de un RDBMS como Oracle, DB2 o SQL Server.

A pesar de que pueden utilizarse índices y relacionar tablas para crear vistas maestro-detalle, lo cierto es que con *MyBase* trabajamos con tablas individuales y no con una base de datos propiamente dicha. Al no existir un servidor, no es posible la existencia de procedimientos almacenados ni el proceso de sentencias SQL.

La estructura de las tablas de datos se define manualmente, utilizando el editor de campos que ya conocemos, aunque también existe la posibilidad de importar los datos de otro conjunto ya existente o, incluso, generar el nuevo conjunto a partir de un documento XML.

Dado que la base de datos se recupera completamente en memoria, no es posible el uso concurrente de la información por varios usuarios. Teóricamente sería posible la edición por separado y una posterior reconciliación, aunque *MyBase* no está pensado para esto.

El único componente que vamos a conocer, llamado `TClientDataSet`, es la base no sólo de *MyBase*, sino también de las aplicaciones cliente/servidor y en varias capas con DBX.

El componente `TClientDataSet`

Encontrará este componente en la página **Data Access** de la Paleta de componentes. Es un derivado de `TCustomClientDataSet` que, a su vez, está derivado de `TDataSet`. Esto significa, por tanto, que cuenta con todos los métodos de cualquier otro derivado de `TDataSet`, algunos de los cuales ha conocido básicamente en capítulos previos. En un capítulo posterior entraremos en mayor detalle sobre el funcionamiento de un `TDataSet`, conociendo los métodos y propiedades que hacen posible la navegación y edición de datos.

No todas las propiedades de `TClientDataSet` son útiles cuando lo que se quiere es desarrollar una aplicación *MyBase*. `RemoteServer`, `ProviderName` o `ConnectionBroker`, por ejemplo, son propiedades que tienen que ver con

el uso de proveedores de datos, ya sean locales o remotos. Tampoco resultarán de interés, en este momento, las propiedades `PacketRecords`, `FetchOnDemand` y `CommandText`.

Si inserta un `TClientDataSet` en un formulario o módulo de datos, verá, en el Inspector de objetos, que no dispone de ninguna propiedad `DatabaseName`, `Connection`, `TableName` o `SQL`. Como se ha dicho, este componente no se conecta a ningún origen de datos para recuperar información, exceptuando los proveedores que en este momento no nos interesan. En contrapartida, y al igual que muchos otros componentes, éste dispone de los métodos `SaveToFile()`, `SaveToStream()`, `LoadFromFile()` y `LoadFromStream()`, permitiendo el almacenamiento y recuperación local de los datos, ya sea directamente en un archivo o en un flujo o *stream*.

Al igual que otros derivados de `TDataSet`, como `TTable` o `TIBTable`, un componente `TClientDataSet` dispone de todas las propiedades relacionadas con columnas e índices, como `Fields`, `FieldDefs`, `IndexDefs` o `IndexName`. Conocerá la mayoría de ellas de inmediato, en los puntos siguientes, usándolas para definir la estructura de la tabla y los índices que desee asociar.

Los métodos de `TClientDataSet` también son similares a los de otros derivados de `TDataSet`, pudiendo crear la tabla, los índices, efectuar tareas de edición, etc. No es necesario, sin embargo, recurrir a esos métodos si lo único que se necesita es crear una simple interfaz de usuario. Mediante un componente `TDataSource`, que también conoce, es posible conectar un `TClientDataSet` con controles corrientes de edición de datos, los alojados en la página **Data Controls** de la Paleta de componentes.

Definición de la estructura de una tabla

Comenzaremos nuestra andadura con *MyBase* definiendo una tabla sencilla, con algunas columnas relativas a información sobre libros. Esta tabla nos servirá, posteriormente, como base para ir extendiendo nuestra base de datos y efectuar otras operaciones, como los enlaces maestro-detalle o el uso de índices para efectuar búsquedas.

Dada la simplicidad inicial, optaremos por insertar el componente `TClientDataSet` directamente en el formulario que aparece por defecto en el proyecto. Éste, desarrollado en Kylix, utiliza la CLX y, por tanto, podrá ser recompilado sin problemas con Delphi 6.

Asignamos un nombre al `TClientDataSet`, modificando la propiedad `Name`, y, desplegando el menú emergente asociado a ese componente, abrimos la ventana del editor de columnas. Pulsamos la combinación **Control-N** para añadir una nueva columna, como se aprecia en la figura 13.1. En ella aparece el formulario con el `TClientDataSet`, el editor de columnas y la ventana donde está definiéndose la primera columna.

El nombre del componente que identificará a cada columna añadida se compone automáticamente, uniendo el nombre del `TClientDataSet` y el de la columna, por lo que tan sólo tendremos que ir facilitando el nombre de cada

columna, su tipo y, en caso necesario, el tamaño. Siempre dejaremos marcada la opción **Data**, ya que vamos a definir, en principio, sólo columnas de datos. Al pulsar el botón **Ok** la columna aparecerá en la ventana del editor de columnas.

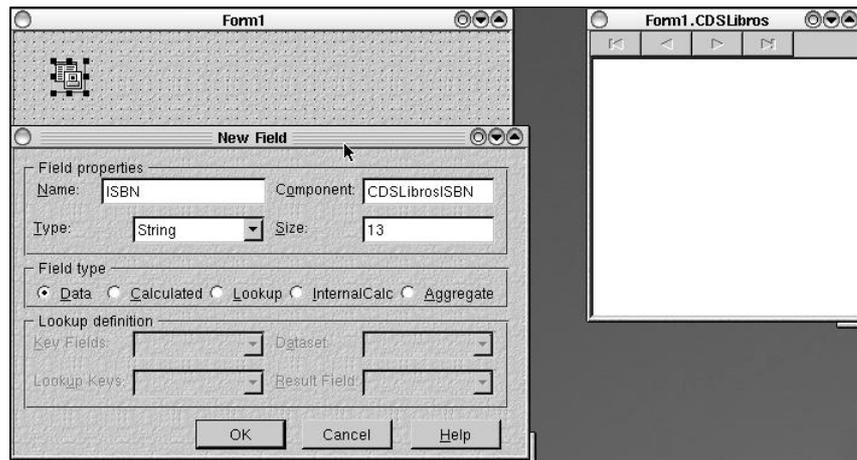


Figura 13.1. Iniciamos el diseño de nuestra primera tabla *MyBase*

Columnas de la tabla

Usaremos esta tabla supuestamente para almacenar información sobre los títulos que tenemos en nuestra biblioteca. Las columnas que añadiremos, sus tipos y longitudes serán los enumerados en la tabla 13.1. Para definir cada columna debe pulsar **Control-N**, introduciendo los datos indicados y pulsando el botón **Ok**.

Tabla 13.1. Columnas del conjunto *CDSLlibros*

Nombre	Tipo	Longitud
ISBN	String	13
Editorial	Smallint	-
Título	String	40
Autor	String	25
Páginas	Smallint	-
Descripción	Memo	-

En la figura 13.2 puede ver, en primer plano, el editor de columnas tras haber definido todas las que se indican. Observe, en segundo plano, el editor de código. En él encontrará una serie de variables, realmente son objetos, que representan a cada una de las columnas. Su tipo, `TStringField`, `TSmallintField` y `TMemoField`, identifican el tipo de información que pueden contener.

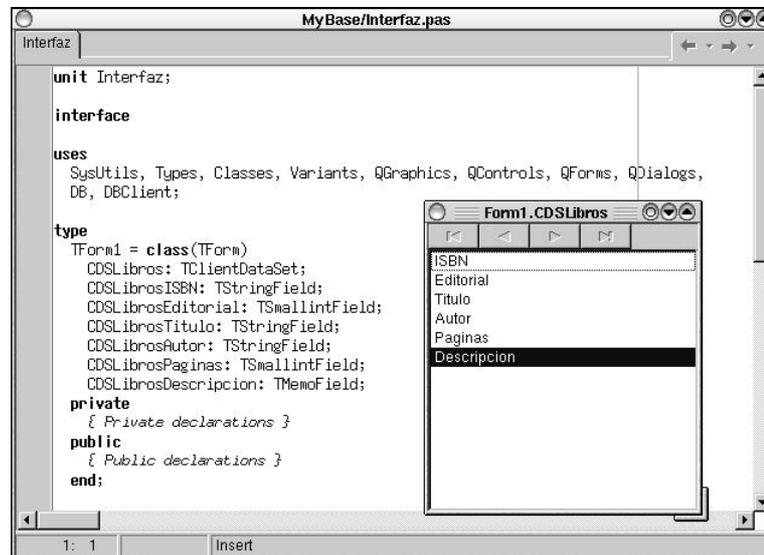


Figura 13.2. Las columnas definidas y los objetos que las representan en nuestro programa

Ahora mismo lo único que tenemos son las definiciones del editor de columnas, que se alojan en el archivo `xfm` como un recurso, y los objetos añadidos a la definición del formulario. En realidad, no existe una tabla de datos, tan sólo un prototipo de su estructura.

Creación de la tabla

En este momento no podríamos usar ninguno de los métodos de `TClientDataSet` para editar y navegar por los datos ya que, como acaba de decirse, la tabla en realidad todavía no existe. Para crearla, pulse el botón secundario del ratón sobre el `TClientDataSet` y seleccione la opción `Create DataSet`. Apparentemente no ocurre nada pero, si se fija, verá que la propiedad `Active` del componente ha tomado el valor `true`.

Otro cambio notable será la lista de opciones disponibles en el menú emergente asociado al componente. Como se aprecia en la figura 13.3, ahora hay múltiples opciones que permiten guardar la tabla en un archivo, ya sea en formato XML, que es lo usual, o bien en formato binario.

Seleccione la opción `Save to MyBase Xml Table` e introduzca como nombre de la tabla `Librox.xml`. Ese archivo será el que conserve la información de nuestra tabla de libros que, en principio, está vacía. No obstante, si abre dicho archivo, por ejemplo en el editor de Delphi 6, verá que dispone de una serie de elementos en los que se describe la estructura de la tabla (véase la figura 13.4). La rama `FIELDS` dispone de un nodo por cada columna, indicando el nombre, tipo y longitud. La última columna es de tipo binario y el subtipo indica que contendrá un texto.

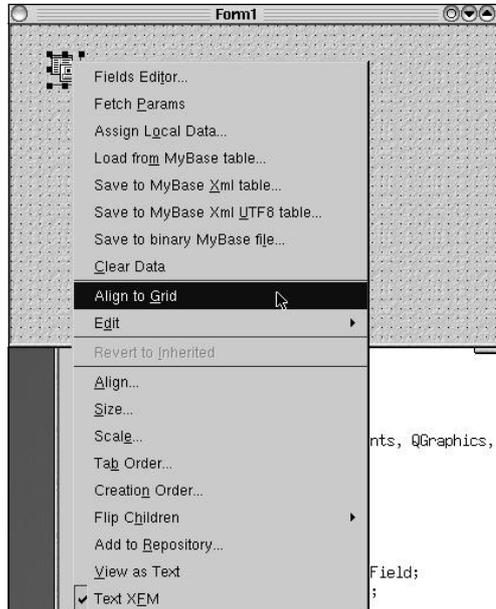


Figura 13.3. Opciones del TClientDataSet tras crear la tabla



Figura 13.4. Aspecto del documento XML generado a partir del TClientDataSet

El elemento ROWDATA, actualmente vacío, será el que contenga la información de la tabla. Conjuntamente, la estructura y los datos conforman lo que se conoce como un paquete de datos o *datapacket*. Después veremos cómo generar este tipo de paquetes de datos partiendo de documentos XML existentes.

Conexión con controles de edición

El trabajo que hemos realizado hasta ahora es, básicamente, de diseño, habiendo definido la estructura de una tabla que, por ahora, será el único elemento de nuestra base de datos. Este trabajo sería equivalente al que, en un capítulo anterior, efectuábamos sirviéndonos del Database Desktop. Como entonces, lo que nos interesa ahora es crear una interfaz de usuario que nos facilite la edición de los datos contenidos en esa tabla.

Partimos del mismo proyecto anterior, en el que teníamos el componente `TClientDataSet` en un formulario y definidas las columnas. Suponiendo que iniciásemos un nuevo proyecto, bastaría con insertar un `TClientDataSet`, desplegar el menú emergente y seleccionar la opción `Load from MyBase table` para recuperar dicha definición.

Nota

Al recuperar la estructura de una tabla previamente definida, usando la citada opción `Load from MyBase table`, el `TClientDataSet` obtiene toda la información que necesita para trabajar con las columnas de la tabla, pero no se crean los objetos que las representan. Para añadir dichos objetos a la definición del formulario, deberá hacer doble clic sobre el `TClientDataSet`, abriendo el editor de columnas, y pulsar la combinación **Control-F**.

Como cualquier otro derivado de `TDataSet`, el componente `TClientDataSet` puede conectarse a un `TDataSource` que, a su vez, servirá como intermediario con los controles de edición y navegación. En la figura 13.5 puede ver como hemos insertado un `TDataSource` y varios `TDBEdit`, un `TDBMemo` y un `TDBNavigator`. Todos ellos se conectan con el `TDataSource` mediante la propiedad `DataSource`. Los controles de edición disponen, además, de la propiedad `DataField` para seleccionar la columna sobre la que actuarán.

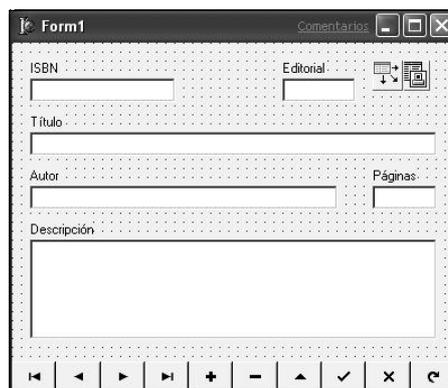


Figura 13.5. Aspecto del formulario para la edición de datos

Si ejecuta el programa tal y como está, comprobará que puede añadir datos, modificarlos, eliminarlos y cambiar de unas filas a otras. Es decir, cuenta con las posibilidades típicas de cualquier programa que opera sobre una base de datos. Pero, al salir del programa y volver a entrar verá que los datos han desaparecido. Es lógico, ya que en ningún momento los almacenamos ni recuperamos.

Almacenamiento de los datos

Como se indicaba anteriormente, el componente `TClientDataSet` cuenta con los métodos `LoadFromFile()` y `SaveToFile()`. Éstos necesitan como único parámetro el nombre del archivo del que va a recuperarse o en el que se almacenará la información. Si, como es habitual, vamos a utilizar siempre el mismo archivo, podemos asignar el nombre a la propiedad `FileName` y, de esta forma, podremos invocar a los dos métodos indicados sin necesidad de facilitar parámetro alguno.

En principio, por tanto, bastaría con asociar el código siguiente a los eventos `OnCreate` y `OnClose` del formulario, de tal forma que los datos se recuperen al iniciar la aplicación y vuelvan a almacenarse al terminar. Hemos asignado el nombre `Libros.xml` a la propiedad `FileName` del `TClientDataSet`, nombre que dimos al archivo al seleccionar la opción `Create DataSet`.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    CDSLibros.LoadFromFile;
end;

procedure TForm1.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    CDSLibros.SaveToFile;
end;
```

Si vuelve a ejecutar el programa verá que ahora sí se conservan los datos que vaya añadiendo. Pruebe a incluir información de un par de libros, posteriormente modifique alguno de esos datos. Al cerrar la aplicación y volver a iniciarla encontrará los datos tal y como los dejó.

Nota

Al asignar un valor a la propiedad `FileName`, facilitando el nombre de un archivo existente, el componente `TClientDataSet` recuperará y guardará automáticamente los datos en ese archivo, sin necesidad de utilizar los métodos `LoadFromFile()` y `SaveToFile()`. La lectura de los datos se efectúa en el momento en que abrimos el `TClientDataSet`, por ejemplo con el método `Open()`, mientras que la escritura se produce automáticamente al liberar el componente.

El registro de cambios

Si tras efectuar algunas tareas de edición echa un vistazo al contenido del archivo XML, quizá le sorprenda ver, como en la figura 13.6, la existencia de entradas relativas a un libro que aparecen varias veces. En el caso concreto de esta figura sólo hay datos de dos libros aunque, en apariencia, parecen existir cinco porque ese es el número de entradas.



```
<?xml version="1.0" standalone="yes" ?>
- <DATAPACKET Version="2.0">
- <METADATA>
- <FIELDS>
  <FIELD attrname="ISBN" fieldtype="string" WIDTH="13" />
  <FIELD attrname="Editorial" fieldtype="i2" />
  <FIELD attrname="Titulo" fieldtype="string" WIDTH="40" />
  <FIELD attrname="Autor" fieldtype="string" WIDTH="25" />
  <FIELD attrname="Paginas" fieldtype="i2" />
  <FIELD attrname="Descripcion" fieldtype="bin.hex" SUBTYPE="Text" />
</FIELDS>
<PARAMS CHANGE_LOG="3 1 8 4 2 8 5 3 8" />
</METADATA>
- <ROWDATA>
<ROW RowState="1" ISBN="84-415-1175-6" Editorial="1" Titulo="Guía práctica de SQL
Server 2000" Autor="Francisco Charte" Paginas="336" Descripcion="Guía de iniciación
a SQL Server 2000" />
<ROW RowState="1" ISBN="84-415-0967-0" Editorial="1" Titulo="Programación con
Delphi 5" Autor="Francisco Charte" Paginas="1182" Descripcion="Libro que trata la
mayoría de los aspectos de uso y programación con Delphi 5" />
<ROW RowState="9" ISBN="84-415-1175-6" Editorial="1" Titulo="Guía práctica de SQL
Server 2000" Autor="Francisco Charte" Paginas="336" Descripcion="Libro breve que
muestra las tareas más habituales del administrador y usuario de SQL Server
2000" />
<ROW RowState="8" ISBN="84-415-0967-0" Editorial="1" Titulo="Programación con
Delphi 5" Autor="Francisco Charte" Paginas="1182" Descripcion="Libro que trata la
mayoría de los aspectos de uso y programación con Delphi 5, desde la creación de
interfases de usuario hasta el desarrollo de componentes distribuidos" />
<ROW RowState="8" ISBN="84-415-1175-6" Editorial="1" Titulo="Guía práctica Microsoft
SQL Server 2000" Autor="Francisco Charte" Paginas="336" />
</ROWDATA>
</DATAPACKET>
```

Figura 13.6. Contenido del archivo XML tras algunas operaciones de edición

Observando cada etiqueta `ROW` notaremos que todas ellas contienen un atributo `RowState` indicando el estado de la fila. Sólo dos de ellas contienen el estado 1, correspondiente a la primera inserción de la fila en la tabla. Las demás indican otros estados comunicando la modificación de una o más de sus filas.

Cuando se recupera una tabla `MyBase`, mediante el método `LoadFromFile()` ya mencionado, toda la información se almacena en la propiedad `Data` utilizando un formato interno del componente `TClientDataSet`. Los cambios que van efectuándose, por ejemplo añadiendo o modificando filas de datos, no modifican el contenido de `Data`, sino que se almacenan separadamente en la propiedad `Delta`.

Mantener independientemente dos bloques de datos, los originales y los modificados, tiene algunas ventajas y es una técnica que, en ocasiones, resulta indispensable, por ejemplo al utilizar los `ClientDataSet` conectados a proveedores de datos, ya que de lo contrario no sería posible la actualización. En nuestro caso, sin embargo, antes de guardar los datos nos interesaría combinar la información de `Data` y `Delta` a fin de obtener una sola versión, consolidada, como la que aparece en la figura 13.7. Para ello, lo único que necesitamos es llamar al método `MergeChangeLog()` antes de utilizar `SaveToFile()`.

Figura 13.7. Aspecto del documento tras combinar los cambios y guardarlos

El mantenimiento de un registro separado de cambios también permite deshacerlos. El método `UndoLastChange()` permite deshacer el último cambio efectuado, ya sea una modificación, una eliminación o una inserción.

Ahorro de trabajo y memoria

Mantener un registro de cambios separado, en lugar de efectuarlos directamente sobre los datos ya existentes, puede resultar útil, e incluso imprescindible como se ha dicho, en caso de que el componente `TClientDataSet` se

utilice como intermediario, en el cliente, para manipular información obtenida desde una base de datos o un servidor de aplicaciones a través de otros componentes.

Utilizando el mencionado componente como estamos haciéndolo ahora, como solución simple de acceso a datos en aplicaciones cliente, ese registro de cambios, no obstante, supone un consumo innecesario de memoria al mantenerse constantemente dos bloques de datos separados: *Data*, con las filas originales, y *Delta* con los cambios. Además, también representa un mayor trabajo para la aplicación, no ya por tener que llamar al método `MergeChangeLog()` sino por la función que éste desempeña. Tendrá que recorrer todo el registro de cambios e ir consolidando con los datos originales para dar lugar a un solo bloque.

La propiedad `LogChanges`, que por defecto tiene el valor `true`, es la que controla la existencia de ese registro de cambios. Si damos el valor `false` a `LogChanges`, todos los cambios efectuados se aplicarán directamente sobre la propiedad *Data*, en lugar de almacenarse en *Delta*.

Como podrá comprobar en el Inspector de objetos, teniendo seleccionado el componente `TClientDataSet`, la propiedad `LogChanges` no es accesible en la fase de diseño. Es necesario, por tanto, efectuar el cambio de valor desde el código del programa.

Cabría esperar que una asignación inmediata, por ejemplo en el evento de creación del formulario, tuviese efecto, pero la realidad es que cualquier llamada posterior al método `LoadFromFile()` devolvería a `LogChanges` su valor original.

Por ello, es necesario efectuar esa asignación siempre detrás de la llamada a `LoadFromFile()`, quedando nuestro método `FormCreate()` de la siguiente forma:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  CDSLibros.LoadFromFile;
  CDSLibros.LogChanges := false; // Sin registro de cambios
end;
```

Tareas de edición

Si bien podemos editar los datos de un `TClientDataSet` con la creación de una interfaz simple, como hemos hecho en el punto anterior, lo cierto es que, como derivado de `TDataSet`, disponemos de todas las propiedades y métodos necesarios para efectuar las tareas habituales en un conjunto de datos: inserción y eliminación de filas, actualización de las existentes, búsquedas basadas en índices, establecimiento de rangos y filtros, etc.

En el capítulo quince conocerá la forma de realizar la mayoría de estas operaciones ya que, como puede suponer, son generales a todos los derivados de `TDataSet`, siendo aplicables no sólo cuando trabajamos con *MyBase* sino también con BDE, DBX, IBX o ADO.

Trabajo con índices

En principio, las filas existentes en un `TClientDataSet` conservan el orden en que han ido añadiéndose. Esto es así porque no existe un índice activo que establezca ese orden. Los índices, además, le servirán para efectuar búsquedas rápidas de datos o bien establecer límites para actuar sobre ciertos rangos de filas.

Una forma rápida de establecer un índice consiste en utilizar la propiedad `IndexFieldNames`, a la que podemos asignar el nombre de una o más columnas. Esto provocará la creación de un índice temporal, en memoria, con unas opciones por defecto, como el orden ascendente o la ausencia de distinción entre mayúsculas y minúsculas. Basta que modifiquemos, incluso en ejecución, el valor de `IndexFieldNames` para crear automáticamente el índice y alterar el orden de las filas.

Otra alternativa, especialmente interesante si deseamos un mayor control sobre el funcionamiento del índice y, además, deseamos que éste se encuentre predefinido, es crearlo en el mismo momento en que se genera el conjunto de datos. Los pasos a dar, asumiendo que acabamos de insertar el `TClientDataSet` o bien hemos elegido la opción `Clear Data` de su menú emergente, serían los siguientes:

- Definir las columnas de la tabla, ya sea con el método descrito previamente o bien haciendo doble clic sobre la propiedad `FieldDefs` para abrir un editor desde el cual podremos establecer las características de cada columna.
- Haga doble clic sobre la propiedad `IndexDefs` para abrir el editor que puede ver en la parte derecha de la figura 13.8. En ella podrá ver dos índices predefinidos. Pulse el primer botón que hay en la parte superior para crear un nuevo índice.
- Establezca el nombre del índice, en la propiedad `Name`, e introduzca en la propiedad `Fields` el nombre de la columna o columnas que formarán parte del índice. En caso de introducir varias columnas, separe los nombres con punto y coma.
- Despliegue la propiedad `Options` y active las opciones que le interesen indicando, por ejemplo, si el índice es el primario, si los valores deben o no repetirse, si se distinguirá o no entre mayúsculas y minúsculas, etc.
- Dependiendo de las opciones anteriores, quizá deba asignar a las propiedades `DescFields` y `CaseInsFields` el nombre de una o más columnas. Éstas deben aparecer también en la propiedad `Fields`, ya que es en ella donde se establece el orden de las columnas en el índice. La propiedad `DescFields` determina las columnas cuyo orden será descendente en vez de ascendente, mientras que `CaseInsFields` indica en qué columnas no se distinguirá entre mayúsculas y minúsculas en caso de que se haya activado esta distinción globalmente para todas las demás columnas.

- Despliegue el menú emergente del `TClientDataSet` y seleccione la opción `Create DataSet`, primero, y `Save to MyBase Xml table`, después. Ya tiene definidos los índices de la tabla.

Durante la ejecución, puede utilizar la propiedad `IndexName` del componente `TClientDataSet` para alternar entre los índices predefinidos.

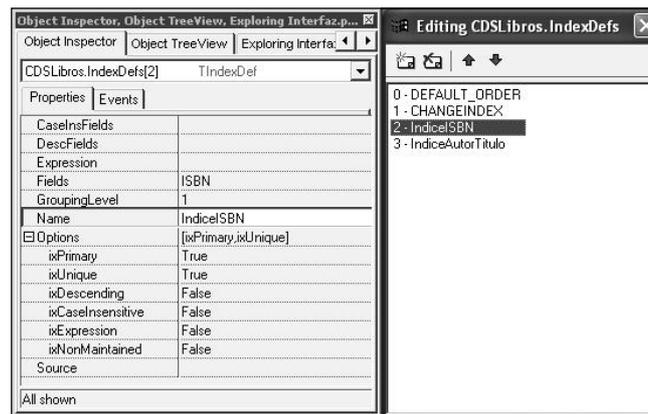


Figura 13.8. Definición de un índice en un `TClientDataSet`

Nota

En un capítulo posterior aprenderá a utilizar estos índices que hemos definido para efectuar búsquedas y otras operaciones. El sistema para crear índices en otras tablas, sean o no *MyBase*, es similar al que acaba de describirse.

Relaciones maestro-detalle

El componente `TClientDataSet` dispone de las propiedades `MasterSource` y `MasterFields`, lo cual facilita la creación de relaciones maestro-detalle entre varias tablas de la misma forma que lo haríamos usando cualquier otro mecanismo de acceso a datos. La diferencia, obviamente, es que todos los datos de ambas tablas se encontrarán en memoria, recuperándose y almacenándose de un archivo en disco.

Lógicamente, para definir una relación de este tipo necesitaremos al menos dos tablas. Por ello, aparte de recuperar en un `TClientDataSet` la tabla de libros previamente definida, vamos a crear una nueva tabla con otro `TClientDataSet`. Su estructura será muy simple, contando tan sólo con dos columnas: `Editorial`, de tipo `Smallint`, y `Nombre`, una cadena de 25 caracteres.

Aparte de los dos `TClientDataSet`, insertaríamos en el formulario también dos `TDataSource` y dos `TDBGrid`, enlazando cada `TDBGrid` con un

TDataSource y éste, a su vez, con un TClientDataSet. En la figura 13.9 puede ver el aspecto que tendría el formulario con todos los componentes.

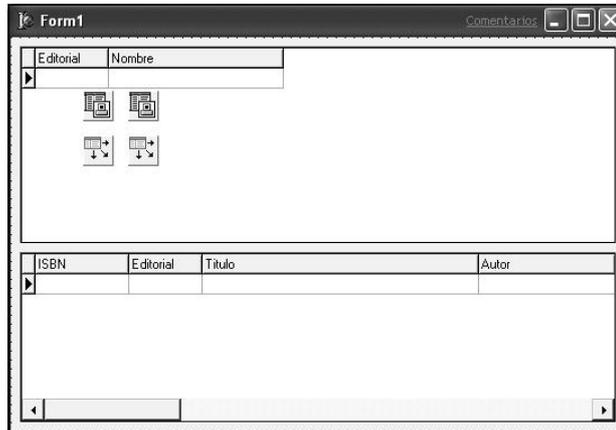


Figura 13.9. Formulario con las rejillas de editoriales y libros

Seleccione el segundo TClientDataSet, correspondiente a la tabla de libros, y asigne a la propiedad MasterSource una referencia al TDataSource de la tabla de editoriales. Haga clic sobre la propiedad MasterFields y, en la ventana que aparece, cree la relación entre ambas tablas, como se ha hecho en la figura 13.10.

Por último, haga doble clic sobre el formulario e introduzca luego el código siguiente:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    CDSEditoriales.Open; // Abrimos las tablas
    CDSEditoriales.LogChanges := false; // y desactivamos el
    CDSLlibros.Open; // control de cambios
    CDSLlibros.LogChanges := false;
end;

```

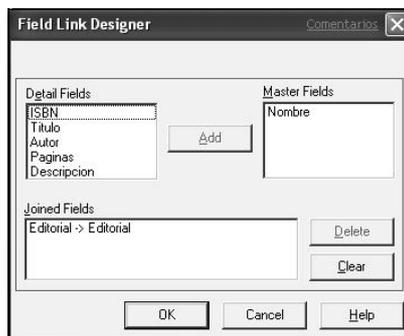


Figura 13.10. Aspecto del formulario con las cuadrículas para mostrar la relación maestro-detalle

Hemos asignado el nombre de los archivos XML a la propiedad `FileName` de cada uno de los `TClientDataSet`, por lo que no es necesario utilizar los métodos `LoadFromFile()` ni `SaveToFile()`, según se indicó previamente. Al ejecutar el programa, podrá ver que en la cuadrícula que hay en la parte inferior aparecen sólo los libros correspondientes a la editorial que se ha seleccionado en la primera cuadrícula. Al introducir los datos de un nuevo libro, el código de la editorial aparece automáticamente.

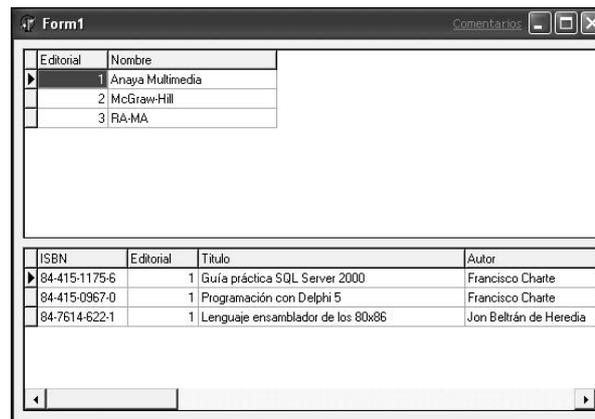


Figura 13.11. Aspecto del programa en funcionamiento, mostrando los libros de la editorial seleccionada

Otras formas de crear las tablas

Hasta ahora hemos creado nuestras tablas *MyBase* utilizando el editor de columnas del componente `TClientDataSet` o bien la propiedad `FieldDefs` del mismo. No son las únicas alternativas y, como se indicaba anteriormente, también es posible crear las tablas partiendo de conjuntos de datos existentes, por ejemplo tablas *InterBase* o *Paradox*.

Otra posibilidad es que tanto la estructura como los datos se encuentren no en una base de datos, que sería lo más habitual, sino en un documento XML que, posiblemente, hayamos recibido de terceros. En este caso, tal y como va a ver inmediatamente, el proceso no es tan directo, puesto que precisaremos una transformación.

Desde otras bases de datos

Un caso común será el que encontremos al convertir nuestras aplicaciones para un solo usuario desarrolladas con versiones previas de Delphi, habitualmente utilizando tablas *Paradox* o *dBase*, para que funcionen tanto en Delphi 6 como en Kylix. La estructura y los datos, que ahora están en ese tipo de tablas, deben ser convertidas generando las tablas *MyBase* que haga falta.

El proceso es el mismo si el origen de datos es InterBase o cualquier otro servidor. Realmente, lo único que necesitamos es un componente que, derivando de `TDataSet`, contenga el conjunto de datos a convertir. Como ejemplo, vamos a tomar la base de datos InterBase, utilizada en un capítulo previo, que contenía información sobre artículos y revistas.

Comenzaríamos incluyendo en el formulario o módulo de datos los componentes necesarios para definir el conjunto de datos a extraer. En este caso concreto, utilizaríamos un `TIBDatabase`, un `TIBTransaction` y un `TIBTable`, asignando los valores apropiados para acceder a la tabla de revistas. En lugar de una tabla completa podríamos usar un componente `TIBQuery` para componer una consulta, como se aprecia en la figura 13.12. En este caso recogemos información de dos tablas distintas.

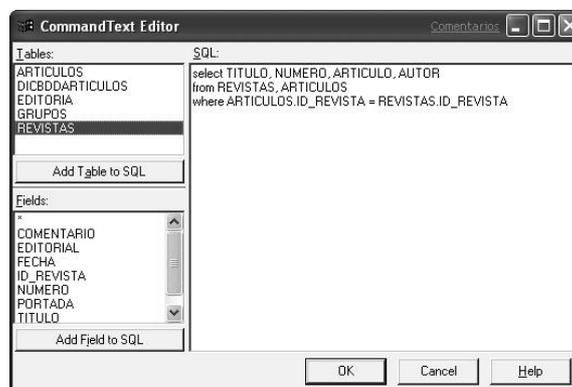


Figura 13.12. Definimos la consulta para la extracción de datos de dos tablas

El paso siguiente sería la inclusión de un `TClientDataSet` que recogerá los datos de la tabla o consulta. Desplegando su menú emergente seleccionaríamos la opción `Assign Local Data`. Ésta hará aparecer una ventana, similar a la de la figura 13.13, en la que podremos elegir uno de los conjuntos de datos existentes en el mismo contenedor. A partir de ese momento el `TClientDataSet` ya dispone de la estructura y los datos, pudiendo usar la opción `Save to MyBase Xml table` para guardarlos y operar sobre ellos según se ha descrito en los puntos precedentes.

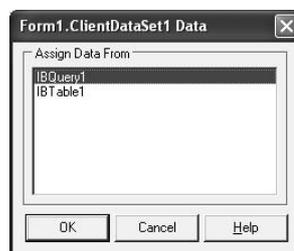


Figura 13.13. Seleccionamos el conjunto de datos a importar al `TClientDataSet`

Desde documentos XML

Muchas veces la información sobre la que es preciso trabajar procede de terceras partes que, usualmente, no utilizarán ni los mismos formatos de bases de datos ni las mismas herramientas que nosotros. Siempre cabe la posibilidad, sin embargo, de que exporten esa información a un documento XML que nosotros, posteriormente, utilizaríamos como punto de partida.

Imaginemos que hemos recibido de esa supuesta tercera empresa uno de los documentos XML usados como ejemplo en el décimo capítulo, conteniendo una lista de productos de software y libros. Nosotros estamos interesados en importar particularmente la lista de datos referente a libros, generando un paquete de datos que podamos usar con un `TClientDataSet`. Seleccione del menú `Tools` de Delphi la opción `XML Mapper`. Esta utilidad permite aplicar transformaciones a un documento XML a fin de generar un paquete de datos. Asumiendo que hemos usado la opción `File>Open` para recuperar el archivo `ListaProductos3.xml`, el aspecto de esta herramienta sería el mostrado en la figura 13.14. Con ella podemos tanto convertir XML en un paquete de datos, que es lo que nos interesa en este momento, como dar el paso inverso.

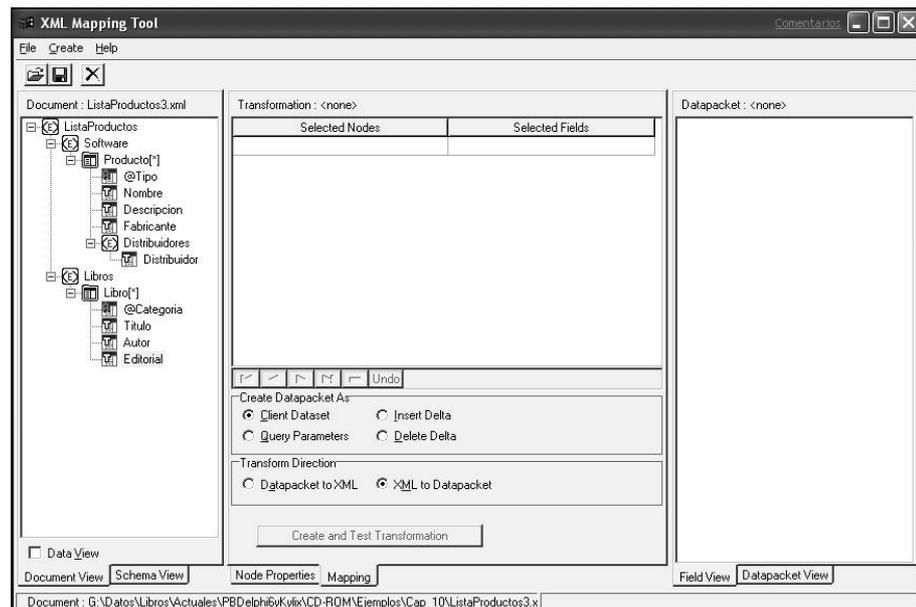


Figura 13.14. Aspecto de la herramienta XML Mapping

Ya que los datos que nos interesan son los relativos a libros, seleccionamos en la lista de la izquierda el elemento `@Categoría` de la rama `Libros` y pulsamos la combinación `Control-L` para añadirlo a la lista de nodos elegidos. Repetimos la operación con los tres elementos que hay debajo: `Título`, `Autor` y `Editorial`.

Teniendo seleccionado cualquiera de los elementos que acaban de añadirse, pulsamos la combinación `Control-D` o bien abrimos el menú emergente y luego

elegimos la opción **Create DataPacket from XML**. El resultado será la aparición, en el panel derecho, de la estructura del paquete de datos a generar. En ella puede ver los atributos de cada columna.

Usando la página **Node Properties**, en la parte central de la ventana, puede acceder a los atributos de cada nodo y modificarlos si fuese necesario. En la figura 13.15, por ejemplo, puede ver cómo definimos el ancho máximo de una de las columnas modificando el valor que aparecía por defecto.

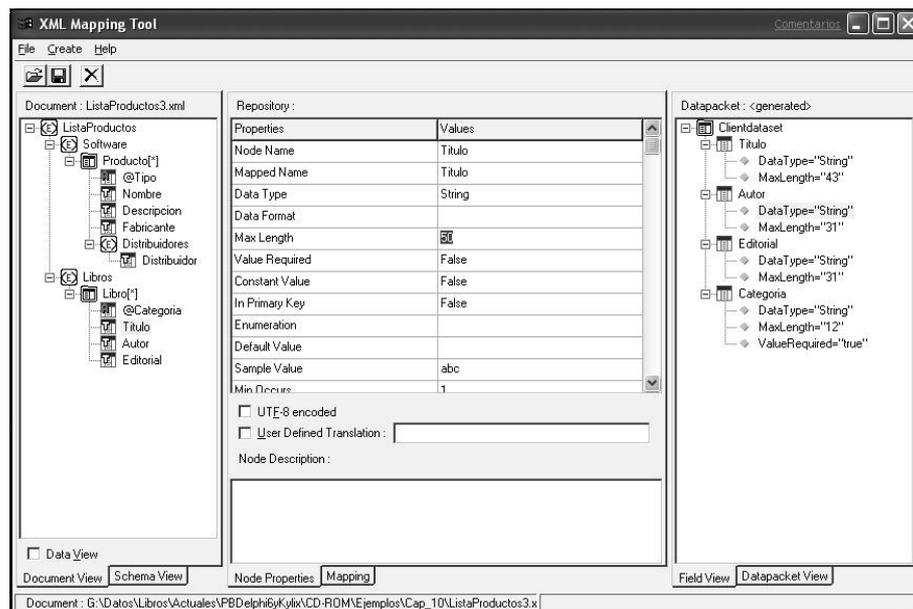


Figura 13.15. Establecemos las propiedades de los nodos

Nota

Si efectúa cambios en las propiedades de los nodos, deberá usar de nuevo la opción **Create DataPacket from XML** para actualizar el paquete de datos que aparece a la derecha.

Dejando las opciones de creación y transformación en su estado por defecto, **Client Dataset** y **XML to Datapacket**, pulsamos el botón **Create and Test Transformation** para ejecutar la transformación de datos y comprobar que el resultado es el que esperamos. Se abrirá una nueva ventana (véase figura 13.16) con los datos obtenidos del documento XML. Pulsando el segundo botón guardamos el paquete de datos en un archivo que, finalmente, será el que recuperemos desde el componente **TClientDataSet** utilizando la opción **Load from Mybase table**. En la figura 13.17 puede ver los datos, que originalmente estaban en un documento XML que habíamos creado con el Bloc de notas o el Editor de código, mostrados en un **TDBGrid** conectado al **TClientDataSet**.

Titulo	Autor	Editorial	Categoria
Programación en Windows 2000	Francisco Charle Djeda	Anaya Multimedia	Programacion
Cómo programar con Visual Basic para torpes	Francisco Charle Djeda	Anaya Multimedia	Programacion
Java 1.2 al descubierto	Jamie Jaworski	Prentice Hall	Programacion
Turbo/Borland Pascal 7	Luis Joyanes Aguilar	McGraw-Hill	Programacion

Figura 13.16. Comprobamos que el resultado de la transformación es el esperado

Categoria	Titulo	Autor	Editorial
Programacion	Programación en Windows 2000	Francisco Charle Djeda	Anaya Multimedia
Programacion	Cómo programar con Visual Basic para torpes	Francisco Charle Djeda	Anaya Multimedia
Programacion	Java 1.2 al descubierto	Jamie Jaworski	Prentice Hall
Programacion	Turbo/Borland Pascal 7	Luis Joyanes Aguilar	McGraw-Hill

Figura 13.17. Los datos en una cuadrícula conectada a un `TClientDataSet`

Teniendo los datos en el `TClientDataSet`, no importa si la estructura la hemos definido nosotros manualmente, ha sido importada desde un derivado de `TDataSet` u obtenida mediante la herramienta XML Mapping. Las operaciones disponibles y la metodología de trabajo sería la misma ya descrita en los puntos previos.

Distribución de las aplicaciones

Una aplicación *MyBase* utiliza como único motor de datos el componente `TClientDataSet`, residiendo parte de su funcionalidad en la librería de enlace

dinámico `midas.dll`, en el caso de Windows, o en el módulo `libmidas.so.1` si se trata de una aplicación Linux. Ése es el único archivo adicional, relacionado con el acceso a datos, que tendremos que redistribuir.

En el caso de Delphi 6, si lo deseamos puede integrarse ese archivo en el ejecutable añadiendo el módulo `midaslib` a la cláusula `uses` del proyecto. De esta forma tendremos la aplicación contenida en un solo archivo, lista para distribuir sin dependencias de acceso a datos.

Resumen

Este capítulo le ha servido para conocer un nuevo mecanismo para desarrollar aplicaciones que utilizan bases de datos pero sin necesidad de conectar a un servidor ni compartir esos datos con otras aplicaciones o usuarios. Hasta cierto punto, *MyBase* puede ser una alternativa al uso de tablas tipo Paradox o dBase, aunque hay casos en que éstas serán más útiles, por ejemplo si es necesario el acceso simultáneo de varios usuarios a los datos y no quiere utilizarse un verdadero RDBMS.

Aunque se han descrito los fundamentos básicos para facilitar la edición de los datos, será en capítulos posteriores donde conozca la mayoría de los controles disponibles y las tareas de edición habituales, tareas que son generales a los distintos mecanismos de acceso a datos, no exclusivas de *MyBase*.

Las aplicaciones del componente `TClientDataSet` no quedan limitadas a su uso en bases de datos *MyBase* sino que, como veremos en el siguiente y en posteriores capítulos, es fundamental para hacer posible la edición trabajando con DBX o la creación de aplicaciones distribuidas con *DataSnap*.