Programación de aplicaciones Delphi con acceso a base de datos

por Francisco Charte

PROGRAMACIÓN DE APLICACIONES DELPHI CON ACCESO A BASES DE DATOS

por Francisco Charte

© 2016 | Danysoft

DERECHOS RESERVADOS

El contenido de esta publicación tiene todos los derechos reservados, por lo que no se puede reproducir, transcribir, transmitir, almacenar en un sistema de recuperación o traducir a otro idioma de ninguna forma o por ningún medio mecánico, manual, electrónico, magnético, químico, óptico, o de otro modo. La persecución de una reproducción no autorizada tiene como consecuencia la cárcel y/o multas.

LIMITACIÓN DE LA RESPONSABILIDAD

Tanto el autor como en Danysoft hemos revisado el texto para evitar cualquier tipo de error, pero no podemos prometerle que el libro esté siempre libre de errores. Por ello le rogamos nos remita por e-mail sus comentarios sobre el libro a attcliente@danysoft.com

DESCUENTOS ESPECIALES

Recuerde que Danysoft ofrece descuentos especiales a centros de formación y en adquisiciones por volumen. Para más detalles, consulte con Danysoft.

MARCAS REGISTRADAS

Todos los productos y marcas se mencionan únicamente con fines de identificación y están registrados por sus respectivas compañías.

Autor: Francisco Charte

Publicado por Danysoft

Avda. de la Industria, 4 Edif. 1

28108 Alcobendas, Madrid. España.

902 123146 — www.danysoft.com

IMPRESO EN ESPAÑA

Depósito Legal | M-1195-2016

© Danysoft | Madrid | 2016 | versión en castellano.

A mi familia

CONTENIDOS EN BREVE

Introducción **39**

PARTE I ACCESO A DATOS LOCALES

Introducción a FireDAC **61**

Herramientas BDD en Delphi **89**

Interfaces de usuario con conexión a datos **123**

InterBase embebido **155**

Bases de datos de escritorio **181**

Bases de datos en memoria **209**

Bases de datos y Unicode **229**

PARTE II APLICACIONES CLIENTE/SERVIDOR

Introducción al desarrollo cliente/servidor **263**

Transacciones, bloqueos y notificación de cambios **311**

Trabajar sin conexión al RDBMS **333**

Interbase **359**

PARTE III APLICACIONES DISTRIBUIDAS

Introducción al desarrollo distribuido con Delphi **375**

Servicios DataSnap **393**

Servicios REST **425**

Introducción a EMS **469**

PARTE IV APÉNDICES

Contenidos

Lista de figuras 23

Agradecimientos 37

Introducción 39

1.1 Aplicaciones móviles 40
1.2 Aplicaciones Windows/OS X 41
1.3 Aplicaciones web 42
1.4 Servicios de acceso a bases de datos y Delphi 43
1.4.1 BDE 44
1.4.2 IBX 47
1.4.3 dbGo 48

1.4.4 dbExpress 50
1.4.5 DataSnap 52
1.4.6 FireDAC 53
1.5 Sobre este libro 55
1.5.1 Estructura 56
1.5.2 Notación 57

PARTE I ACCESO A DATOS LOCALES

Introducción a FireDAC 61

2.1 Hola FireDAC 62
2.1.1 Inicio del proyecto 62
2.1.2 Adición de un módulo de datos 64
2.1.3 Configuración de la conexión 65
2.1.4 Creación de una tabla 67
2.1.5 Introducción de datos 69
2.1.6 Otros componentes a añadir al módulo de datos 71
2.1.7 Diseño de la interfaz de usuario 72
2.1.8 Enlace entre interfaz y datos 73
2.1.9 Prueba de la aplicación 75
2.2 Bases de datos locales y FireDAC 76
2.2.1 InterBase Lite e InterBase ToGo 77
2.2.2 SQLite 78
2.2.3 Microsoft Access 79
2.2.4 Archivos de datos en otros formatos 80
2.3 Estructura de una aplicación que usa FireDAC 81
2.3.1 Controladores FireDAC 83
2.3.2 Conjuntos de datos 85
2.3.3 Elementos de interfaz 86
2.3.4 Otros componentes FireDAC 87
2.4 Resumen 88

Herramientas del Entorno de Delphi para BDD 89

3.1 Módulos de datos 90
3.2 Data Explorer 91
3.2.1 Definición de conexiones 91
3.2.2 Exploración de datos 94
3.2.3 Inserción de componentes de conexión 95
3.3 FireDAC Explorer 97
3.3.1 Archivos de definición de conexiones locales a un proyecto 99
3.3.2 Estructura de un archivo de definición de conexiones FireDAC 100
3.4 FireDAC Monitor 101
3.4.1 Componentes de supervisión FireDAC 101
3.4.2 La interfaz de FireDAC Monitor 103
3.5 Herramientas asociadas a componentes FireDAC 104
3.5.1 El Editor de campos 104
3.5.2 El Editor de consultas 108
3.6 El asistente de Live Bindings 110
3.7 En la práctica 111
3.7.1 Conexión y selección de datos 112
3.7.2 Configuración para la inspección de actividad 113
3.7.3 Edición de la consulta de selección de datos 113
3.7.4 Adición de un campo calculado 114
3.7.5 Diseño de la interfaz de usuario 115
3.7.6 Conexión entre interfaz y datos 116
3.7.7 Código de activación y filtrado 118
3.7.8 Probando la aplicación 118
3.7.9 Preparar la aplicación para su despliegue 119
3.8 Resumen 121

Interfaces de usuario con conexión a datos 123

4.1 VCL versus FMX 124

4.1.1 Plataformas objetivo 124
4.1.2 Hardware y software de base 126
4.1.3 Compatibilidad hacia atrás 128
4.2 FireDAC y aplicaciones VCL 129
4.2.1 Módulo de datos y componentes FireDAC 130
4.2.2 Interfaz de usuario 132
4.3 Componentes VCL con conexión a datos 133
4.4 LiveBindings 135
4.4.1 Componentes LiveBindings esenciales 136
4.4.2 Adición y enlace de controles 140
4.5 Diseño de una interfaz con controles simples y navegación 141
4.5.1 Origen de los datos 142
4.5.2 Diseño de la interfaz de usuario 143
4.5.3 Controles de navegación 144
4.6 Otras configuraciones de datos 145
4.6.1 Relaciones maestro/detalle 146
4.6.2 Campos de búsqueda 147
4.6.3 En la práctica 148
4.7 Resumen 153

Aplicaciones con InterBase embebido 155

5.1 InterBase ToGo 156
5.1.1 Licencias ToGo e IBLite 157
5.2 Herramientas InterBase para desarrollo 157
5.2.1 InterBase Manager 158
5.2.2 La consola de InterBase 159
5.3 InterBase y FireDAC 161
5.4 Caso práctico 162
5.4.1 Configuración de la conexión 163
5.4.2 Selección de datos 164
5.4.3 Diseño de la interfaz 165
5.4.4 Enlace con los datos 167

5.4.5 Gestión de los eventos 169
5.4.6 Comprobación de la aplicación 170
5.5 Configuración de despliegue 170
5.5.1 Ruta de la base de datos 171
5.5.2 Redistribución de IBLite y su licencia 173
5.5.3 Distribución de la base de datos 174
5.5.4 Configuración de permisos 174
5.5.5 Comprobación de la aplicación 175
5.6 Resumen 179

Delphi y Bases de Datos de Escritorio 181

6.1 Acceso a datos en archivos de Microsoft Excel 182
6.1.1 FireDAC y Excel 182
6.1.2 Configuración de FireDAC para usar un controlador ODBC 183
6.1.3 Consultas sobre un libro Excel 184
6.1.4 Supuesto práctico 185
6.1.5 Configuración de la conexión por ODBC 187
6.1.6 Consulta de recuperación de datos 188
6.1.7 Diseño de la interfaz de usuario 189
6.1.8 Apertura de la conexión 191
6.1.9 Exportación de los datos 192
6.2 Uso de datos almacenados en archivos de texto 195
6.2.1 Lectura de archivos CSV con FireDAC 195
6.2.2 Procesamiento local de consultas SQL 197
6.2.3 Consultas sobre archivos CSV 199
6.2.4 Componentes de conexión y procesamiento de consultas 199
6.2.5 Lectura de los datos CSV 200
6.2.6 Diseño de la interfaz 204
6.3 Resumen 206

Bases de datos en memoria con FireDAC 209

7.1 El motor local de SQL de FireDAC 210
7.2 Conjuntos de datos en memoria 211
7.3 Definición de estructuras de datos en memoria 212
7.3.1 En la fase de diseño 213
7.3.2 Durante la ejecución 216
7.4 Caso práctico 219
7.4.1 Configuración de los componentes de datos 220
7.4.2 Diseño de la interfaz de usuario 222
7.5 Exportación de los datos 225
7.5.1 El componente `TFDSQLiteBackup` 226
7.5.2 En la práctica 227
7.6 Resumen 228

Bases de datos y Unicode 229

8.1 Introducción a Unicode 230
8.1.1 Estándares de codificación: ASCII 230
8.1.2 Unicode y las codificaciones UTF-N 232
8.1.3 Unicode y cabecera BOM 235
8.2 Soporte Unicode en Delphi 236
8.2.1 Tipos de cadenas y caracteres 238
8.2.2 Conversión entre codificaciones 241
8.2.3 Lectura/Escritura de archivos Unicode 245
8.3 Bases de datos y Unicode 252
8.3.1 Configuración Unicode en FireDAC 252
8.3.2 Aspectos específicos de los RDBMS 257
8.4 Resumen 259

PARTE II APLICACIONES CLIENTE/SERVIDOR

Introducción al desarrollo cliente/servidor 263

9.1 Conexión con el servidor de datos 264
9.1.1 Ruta y nombre de la base de datos 264

9.1.2 Autenticación de acceso 265
9.1.3 Reutilización de conexiones 267
9.1.4 Gestión de las conexiones durante la ejecución 270
9.2 Obtención de datos 272
9.2.1 Selección de columnas 273
9.2.2 Filtrado de filas 274
9.2.3 Paginación de resultados 275
9.3 Edición de datos 281
9.3.1 Cómo actualiza FireDAC los datos 282
9.3.2 Generación de las sentencias para actualizar datos 284
9.4 Procesamiento de datos en el servidor 296
9.4.1 Agrupación y agregado de datos 296
9.4.2 Otros cálculos sobre los datos 298
9.4.3 Actualización de datos por lotes 301
9.5 Ejecución de scripts SQL 302
9.5.1 Vías para aportar el código SQL a ejecutar 303
9.5.2 Edición y prueba del guión en la fase de diseño 304
9.5.3 Asociar un cuadro de diálogo específico 306
9.5.4 En la práctica 306
9.6 Resumen 309

Transacciones, bloqueos y notificación de cambios 311

10.1 Introducción al uso de transacciones 312
10.1.1 Propiedades de una transacción 312
10.1.2 Operaciones de control de una transacción 313
10.2 Transacciones de base de datos vs transacciones FireDAC 314
10.3 Control de transacciones en FireDAC 315
10.3.1 Transacciones automáticas 315
10.3.2 Transacciones manuales 317
10.3.3 Configuración de las transacciones 318
10.3.4 Transacciones anidadas 319
10.3.5 Conflictos de concurrencia 320

10.4 Notificación de cambios 323
10.4.1 Configuración de la solicitud 324
10.4.2 Registro con el servidor y eventos 325
10.4.3 Actualización automática de datos 326
10.4.4 Transacciones y notificaciones en la práctica 327
10.5 Resumen 331

Trabajar sin conexión al RDBMS 333

11.1 El modo desconectado de FireDAC 334
11.1.1 Conectar/Desconectar de la base de datos 335
11.1.2 Reconexión automática 336
11.2 Actualizaciones por lotes 340
11.2.1 Activación de la actualización por lotes 340
11.2.2 Deshacer cambios hechos a los datos 341
11.2.3 Envío de los cambios pendientes 342
11.2.4 Gestión de conflictos 349
11.2.5 Actualización por lotes centralizada 355
11.3 Resumen 357

Interbase 359

12.1 Versiones de InterBase 360
12.2 Configuración del cliente 361
12.3 Configuración de la conexión 363
12.3.1 Protocolo, servidor y puerto 364
12.3.2 Autenticación 365
12.3.3 Identificación de la base de datos 366
12.3.4 Otros parámetros 367
12.4 Componentes FireDAC Services 368
12.4.1 Metodología general de uso 370
12.4.2 Funcionalidad de los componentes de servicio 371
12.5 Resumen 372

PARTE III APLICACIONES DISTRIBUIDAS

Introducción al desarrollo distribuido con Delphi 375

13.1 Estructura de una aplicación distribuida 376
13.1.1 Servidor de datos 377
13.1.2 Servidor de aplicaciones 378
13.1.3 Interfaz de usuario 378
13.2 Comunicación entre interfaz y servidor de aplicaciones 379
13.3 Introducción a REST 380
13.3.1 Composición de las rutas de acceso 381
13.3.2 Verbos HTTP y operaciones 382
13.4 Introducción a JSON 383
13.4.1 Sintaxis de JSON 383
13.4.2 Trabajar con JSON en Delphi 386
13.5 Resumen 392

Servicios DataSnap 393

14.1 Aspectos fundamentales de DataSnap 394
14.1.1 Aplicación contenedora 394
14.1.2 Elementos del servidor DataSnap 395
14.1.3 Componentes en el servidor 396
14.2 Un servicio simple 398
14.2.1 Implementación del servicio 402
14.2.2 Acceso al servicio desde el IDE 404
14.2.3 Un consumidor simple 406
14.2.4 Control del estado del servidor 413
14.3 DataSnap y conjuntos de datos 415
14.3.1 Aspectos específicos en el servidor 416
14.3.2 Devolución de un conjunto de datos 418
14.3.3 Recepción de los datos en el cliente 419
14.3.4 Devolución de los cambios 422

14.4 Resumen 423

Servicios REST 425

15.1 El asistente DataSnap REST 426
15.1.1 Módulo de clases del servidor 428
15.1.2 Módulo web 429
15.1.3 Módulo de interfaz 435
15.2 Un servicio REST simple 435
15.2.1 Comprobación del servidor desde el navegador 438
15.3 Acceso a servicios REST desde Delphi 440
15.3.1 Componentes de un cliente REST 440
15.3.2 Consumo del servicio simple desde un cliente Delphi 442
15.3.3 El depurador REST 445
15.4 Acceso a servicios DataSnap con jQuery 448
15.4.1 Módulos proxy 448
15.4.2 Una interfaz jQuery Mobile 450
15.4.3 Invocaciones mediante AJAX 453
15.4.4 Integración del cliente JavaScript en el servidor 455
15.5 DataSnap REST y conjuntos de datos 458
15.5.1 FireDAC JSON Reflection 459
15.5.2 Desarrollo del servidor 460
15.5.3 Desarrollo del cliente 463
15.6 Resumen 468

Introducción a EMS 469

16.1 Características de EMS 470
16.1.1 Funcionalidad de EMS 470
16.1.2 Herramientas de desarrollo 471
16.1.3 Componentes para aplicaciones 472
16.2 El servidor EMS 474
16.2.1 Configuración del servidor EMS 474
16.2.2 Consola de control del servidor 476

16.2.3 La consola de desarrollo 476
16.3 Servicios ofrecidos por EMS 478
16.3.1 Gestión de grupos 479
16.3.2 Gestión de usuarios 481
16.3.3 Otros servicios 484
16.4 Componentes Delphi para operar con EMS 484
16.5 Desarrollo de paquetes EMS 489
16.5.1 El asistente para paquetes EMS 490
16.5.2 Atributos y registro de las clases de recurso 490
16.5.3 Versión EMS del servidor de números aleatorios 492
16.6 Resumen 495

PARTE IV APÉNDICES

A El entorno de Delphi 499

A.1 El IDE del Delphi 500
A.1.1 Cómo cambiar la distribución de los paneles del IDE 502
A.1.2 La herramienta IDE Insight 505
A.2 Gestión de proyectos 506
A.2.1 Proyectos y plantillas 506
A.2.2 El Gestor de proyectos 509
A.3 Vistas de diseño 511
A.3.1 Cambiar la vista activa 512
A.3.2 Orientación y máscara 513
A.3.3 Vista maestra y vistas específicas 514
A.3.4 Visualización preliminar de la interfaz 515
A.4 Trabajar con componentes 517
A.4.1 Adición de nuevos componentes 518
A.4.2 Manipulación de componentes 520
A.4.3 Edición de propiedades 521
A.5 Edición de código 524
A.5.1 Enlace entre eventos y métodos 525

A.5.2 Asistencia a la edición de código 526
A.6 Gestión de plataformas 530
A.7 Generación del proyecto 532
A.7.1 Compilación 532
A.7.2 Ejecución y depuración 533
A.8 Resumen 541

B El lenguaje Delphi 543

B.1 Sintaxis básica 544
B.1.1 Módulos de código Delphi 545
B.1.2 Estructura del módulo de programa 546
B.1.3 Estructura de un módulo de código estándar 548
B.1.4 Referencias a módulos 549
B.1.5 Comentarios 552
B.2 Tipos de datos fundamentales 553
B.2.1 Números enteros y de coma flotante 554
B.2.2 Caracteres y cadenas de caracteres 556
B.2.3 Otros tipos de datos básicos 557
B.2.4 Notación para literales e identificadores 558
B.2.5 Enumeraciones 560
B.2.6 Subrangos 563
B.2.7 Vectores y matrices (*Arrays*) 565
B.2.8 Registros 570
B.2.9 Conjuntos 574
B.3 Expresiones 576
B.3.1 Expresiones aritméticas, relacionales y lógicas 576
B.3.2 Expresiones con conjuntos 578
B.3.3 Expresiones con punteros 578
B.3.4 Otros tipos de expresiones 578
B.4 Sentencias 580
B.4.1 Estructuras condicionales 580
B.4.2 Estructuras iterativas 582
B.4.3 Procedimientos y funciones 584

B.4.4 Control estructurado de excepciones 589
B.4.5 Otras sentencias 591
B.5 Clases y sus miembros 592
B.5.1 Cómo definir una nueva clase 593
B.5.2 Visibilidad de los miembros de una clase 595
B.5.3 Construcción de objetos 596
B.5.4 La clase TCollectible 598
B.5.5 La clase TComputer 608
B.6 Miembros de clase y el objeto self 612
B.6.1 La clase TCollection 613
B.7 Métodos anónimos 617
B.7.1 Escenario de uso de un método anónimo 618
B.7.2 Definición de tipo de un método anónimo 619
B.7.3 Cómo pasar métodos anónimos como parámetros 620
B.8 Tipos genéricos 621
B.8.1 Cómo definir un tipo genérico 622
B.8.2 Instanciación y uso de tipos genéricos 624
B.9 Ayudas a la escritura de código 626
B.9.1 Plantillas de código 626
B.9.2 Dar formato al código 629
B.9.3 Navegar por el código 629
B.9.4 Refactorización del código 630
B.9.5 Gestión de versiones del código 631
B.10 Resumen 632

C Integración de Delphi con Git 633

C.1 Instalación de Git 635
C.1.1 Uso de Git en Windows 636
C.1.2 Configuración de Git 637
C.2 Integración de Git con Delphi 639
C.3 Operaciones comunes al trabajar con Git y Delphi 640
C.3.1 Clonación de un repositorio Git 640
C.3.2 Inicializar un repositorio local 644

C.3.3 Comprobar el estado de nuestra versión local 646
C.3.4 Confirmar y revertir cambios en un módulo 650
C.3.5 Enviar/Recibir cambios de un repositorio remoto 651
C.4 Configurar un repositorio Git 653
C.5 Resumen 655

D Migración de aplicaciones BDE a FireDac 657

D.1 Instalar BDE en XE7 y versiones posteriores 658
D.2 Diferencias entre BDE y FireDAC 662
D.2.1 Orígenes de datos 662
D.2.2 API de programación 663
D.2.3 Componentes 663
D.2.4 Definición de conexiones 665
D.3 Herramientas de conversión 666
D.4 Resumen 668

Índice alfabético 669

LISTA DE FIGURAS

1.1	Arquitectura de una solución basada en BDE.	45
1.2	Arquitectura de una solución basada en IBX.	48
1.3	Arquitectura de una solución basada en dbGo.	49
1.4	Arquitectura de una solución basada en dbExpress.	51
1.5	Arquitectura de una solución basada en DataSnap.	52
1.6	Arquitectura de una solución basada en FireDAC.	54
2.1	Iniciamos un nuevo proyecto de aplicación multi-dispositivo.	63
2.2	Añadimos un módulo de datos al proyecto.	64
2.3	Seleccionamos el tipo de base de datos.	65
2.4	Abrimos el editor de parámetros de conexión.	66

2.5	INTRODUCIMOS LA SENTENCIA SQL PARA CREAR LA TABLA.	68
2.6	ABRIMOS LA TABLA DESDE EL EXPLORADOR DE DATOS.	70
2.7	INTRODUCIMOS ALGUNOS DATOS EN LA TABLA.	70
2.8	COMPONENTES INTRODUCIDOS EN EL MÓDULO DE DATOS.	71
2.9	CONFIGURAMOS EL TLISTVIEW.	72
2.10	AGREGAMOS AL FORMULARIO UNA REFERENCIA AL MÓDULO DE DATOS.	73
2.11	ENLAZAMOS COLUMNAS DE LA TABLA CON PROPIEDADES DEL CONTROL.	74
2.12	CONFIGURACIÓN FINAL DE LOS ENLACES.	75
2.13	LA APLICACIÓN EN FUNCIONAMIENTO.	76
2.14	ESTRUCTURA DE UNA APLICACIÓN DELPHI QUE USA FIREDAC PARA ACCEDER A UNA BASE DE DATOS LOCAL.	82
2.15	COMPONENTES PARA ENLAZAR LOS CONTROLADORES EN EL PROYECTO.	84
2.16	COMPONENTES FIREDAC ASOCIADOS A ELEMENTOS DE LA INTERFAZ DE USUARIO.	86
2.17	SELECCIÓN DEL PROVEEDOR DE INTERFAZ DE USUARIO.	87
3.1	AGREGAMOS UNA NUEVA CONEXIÓN ASOCIADA AL CONTROLADOR DE INTERBASE.	92
3.2	OPCIONES DE CONFIGURACIÓN DE LA CONEXIÓN.	93
3.3	EXPLORACIÓN DE LA ESTRUCTURA DE UNA BASE DE DATOS.	94
3.4	COMPONENTES AGREGADOS MEDIANTE ARRASTRAR Y SOLTAR.	96
3.5	FIREDAC EXPLORER.	97
3.6	ES POSIBLE CREAR GRUPOS, ORDENAR Y FILTRAR LOS DATOS.	98
3.7	CONFIGURACIÓN DE SUPERVISIÓN EN EL COMPONENTE TFDCONNECTION.	102
3.8	INTERFAZ DE USUARIO DEL PROGRAMA FIREDAC MONITOR.	103
3.9	OPCIONES ASOCIADAS AL COMPONENTE TFDQUERY.	104
3.10	EL EDITOR DE CAMPOS Y SU MENÚ CONTEXTUAL.	105
3.11	PROPIEDADES DE UN OBJETO TFIELD EN EL INSPECTOR DE OBJETOS.	106
3.12	DEFINICIÓN DEL NUEVO CAMPO AGREGADO AL CONJUNTO DE DATOS.	107

3.13	El Editor de consultas nos permite probar las sentencias SQL.	108
3.14	Configuración de parámetros usados en la consulta.	110
3.15	Asistente para Live Bindings.	111
3.16	Formulario finalizada la inserción de componentes.	116
3.17	El programa en funcionamiento.	119
3.18	Inspeccionamos la operación de actualización.	120
4.1	VCL solo está disponible para Win32 y Win64.	125
4.2	FMX también está disponible para OS X, iOS y Android.	126
4.3	Propiedades para aplicar transformaciones en FMX.	127
4.4	Componentes específicos para Windows en la VCL.	128
4.5	Creamos un nuevo proyecto basado en la VCL.	130
4.6	Agregamos al módulo de datos los componentes FireDAC.	131
4.7	Interfaz de usuario con cuadrícula y `TCheckBox`.	132
4.8	Controles VCL con capacidad para conectarse a datos.	134
4.9	Componentes fundamentales de LiveBindings.	136
4.10	Configuración de un `TBindDataSourceDB`.	138
4.11	Diseñador asociado al componente `TBindingsList`.	139
4.12	Lista de clases de enlaces *LiveBindings*.	139
4.13	Insertar y enlazar un nuevo control a un campo de datos.	141
4.14	Podemos seleccionar las columnas a incluir.	143
4.15	Disposición de los controles y enlaces al origen de datos.	144
4.16	La aplicación VCL en funcionamiento.	145
4.17	Establecemos el vínculo entre tabla maestra y de detalle.	149
4.18	Selección de los campos para el vínculo maestro/detalle.	150
4.19	Configuración del campo de búsqueda.	151
4.20	Enlaces entre columnas de datos y controles de interfaz.	152
4.21	El programa con la interfaz maestro/detalle en funcionamiento.	153
5.1	Interfaz del InterBase Manager.	158

5.2	CONSOLA DE INTERBASE.	160
5.3	CONFIGURACIÓN DE CONEXIÓN A LA BASE DE DATOS.	163
5.4	COMPONENTES FIREDAC AGREGADOS AL MÓDULO DE DATOS.	166
5.5	ESTRUCTURA DE LA INTERFAZ DE USUARIO.	167
5.6	CONFIGURACIÓN DE ENLACE A DATOS DEL FORMULARIO.	168
5.7	EL PROGRAMA EN FUNCIONAMIENTO EN WINDOWS.	171
5.8	COMPONENTES REDISTRIBUIBLES A AGREGAR AL PROYECTO.	173
5.9	INCLUIMOS LA PROPIA BASE DE DATOS ENTRE LOS ARCHIVOS A DISTRIBUIR.	174
5.10	CONFIGURACIÓN DE PERMISOS DE LA APLICACIÓN.	175
5.11	ELEGIMOS LA PLATAFORMA EN QUE SE DESPLEGARÁ EL PROYECTO.	176
5.12	PASERVER MANAGER.	176
5.13	LA APLICACIÓN FUNCIONANDO EN EL EMULADOR DE IOS.	177
5.14	ERROR AL INTENTAR CONECTAR.	178
5.15	COMPROBAMOS LOS ARCHIVOS EN LA CARPETA DE DESPLIEGUE.	178
6.1	CONTENIDO DE LA HOJA EXCEL.	186
6.2	CONFIGURACIÓN DE LA CONEXIÓN AL ARCHIVO EXCEL.	187
6.3	CONSULTA PARA OBTENER LOS DATOS DE LA HOJA.	189
6.4	ENLACE ENTRE LA CUADRÍCULA Y LOS DATOS.	190
6.5	CONFIGURACIÓN DE LAS COLUMNAS DEL TGRID.	190
6.6	LA CUADRÍCULA MOSTRANDO LOS DATOS DE LA HOJA EXCEL.	191
6.7	LA APLICACIÓN EN FUNCIONAMIENTO.	193
6.8	LA HOJA DE DATOS AGREGADA POR LA APLICACIÓN DELPHI AL LIBRO EXCEL.	194
6.9	COMPONENTES FIREDAC PARA TRANSFERIR DATOS ENTRE ORIGEN Y DESTINO.	196
6.10	PROCESO DE TRANSFERENCIA DE DATOS CON TBATCHMOVE.	197
6.11	COMPONENTES A INTRODUCIR EN EL MÓDULO DE DATOS.	201
6.12	CONFIGURACIÓN DEL TFDBATCHMOVE.	201
6.13	CONFIGURACIÓN DEL COMPONENTE TFDBATCHMOVETEXTREADER.	202
6.14	ASPECTO DE LA INTERFAZ EN EL DISEÑADOR DE FORMULARIOS.	205

6.15	EL PROGRAMA MOSTRANDO DATOS DE DOS ARCHIVOS CSV.	205
7.1	OPCIONES EN EL MENÚ CONTEXTUAL DE UN TFDMEMTABLE.	211
7.2	DEFINICIÓN DE COLUMNAS DE UNA TABLA CON EL EDITOR DE CAMPOS.	213
7.3	OPCIONES EN EL TFDMEMTABLE TRAS HABER GENERADO EL CONJUNTO DE DATOS.	214
7.4	ESTRUCTURA DE LA INTERFAZ DE USUARIO.	223
7.5	ENLACE DE LAS CUADRÍCULAS CON LOS COMPONENTES DE DATOS.	224
7.6	LA APLICACIÓN MOSTRANDO LA BASE DE DATOS EN MEMORIA.	225
7.7	EXAMINAMOS LOS DATOS EXPORTADOS CON TFDSQLITEBACKUP.	228
8.1	CONJUNTO DE CARACTERES ASCII DE 7 BITS.	231
8.2	CONJUNTO DE CARACTERES ASCII EXTENDIDO DE 8 BITS.	232
8.3	CONJUNTOS DE CARACTERES EXISTENTES EN UNICODE.	233
8.4	OPCIONES DE CODIFICACIÓN DEL BLOC DE NOTAS DE WINDOWS.	236
8.5	COMPROBAMOS EL BOM AÑADIDO POR EL BLOC DE NOTAS.	236
8.6	UN IDENTIFICADOR DE VARIABLE EN JAPONÉS.	237
8.7	DIFERENCIAS ENTRE UNA CADENA UNICODE Y UNA ANSI.	240
8.8	USO DE CADENAS ANSI CON PÁGINAS DE CÓDIGOS ALTERNATIVAS.	242
8.9	CONTENIDO REAL DE CADA VARIABLE.	245
8.10	GUI DEL PROGRAMA PARA CARGAR/GUARDAR CON CODIFICACIÓN.	247
8.11	EL PROGRAMA EN FUNCIONAMIENTO MOSTRANDO LA CONFIGURACIÓN DE UN ARCHIVO RECIÉN ABIERTO.	251
8.12	VISTA PARCIAL DEL CONTENIDO DEL ARCHIVO CSV.	254
8.13	DEFINICIÓN DE LOS CAMPOS DEL ARCHIVO CSV.	255
8.14	LA APLICACIÓN MOSTRANDO LOS DATOS LEÍDOS DEL ARCHIVO CSV.	257
8.15	CONFIGURACIÓN DE LA CODIFICACIÓN PARA UNA CONEXIÓN A INTERBASE.	258
9.1	CONFIGURACIÓN DEL MÉTODO DE AUTENTICACIÓN.	267
9.2	LISTA DE CONEXIONES Y PARÁMETROS.	272
9.3	SELECCIÓN DE CAMPOS CONCRETOS EN UNA CONSULTA.	273
9.4	CONSULTA CON FILTRADO DE FILAS Y SELECCIÓN DE COLUMNAS.	275

9.5	ELEMENTOS A INTRODUCIR EN EL FORMULARIO.	278
9.6	EL PROGRAMA EN FUNCIONAMIENTO MOSTRANDO UNA PÁGINA DE DATOS.	281
9.7	OPCIONES DE ACTUALIZACIÓN DE UN CONJUNTO DE DATOS.	283
9.8	AJUSTAMOS LA CONSULTA A EJECUTAR.	285
9.9	SE LOCALIZA LA FILA A ACTUALIZAR USANDO LA CLAVE PRIMARIA.	286
9.10	LA CLÁUSULA WHERE USA LA CLAVE PRIMARIA Y LOS CAMPOS MODIFICADOS.	287
9.11	LA SENTENCIA DE ACTUALIZACIÓN PODRÍA AFECTAR A MÚLTIPLES FILAS.	288
9.12	CONSULTA CON RELACIÓN ENTRE DOS TABLAS.	289
9.13	LA CLAVE PRIMARIA EMPLEADA NO ES LA CORRECTA.	290
9.14	ESTABLECEMOS LA CLAVE PARA ACTUALIZAR.	291
9.15	LA ACTUALIZACIÓN SE COMPLETA SIN PROBLEMAS.	292
9.16	INTRODUCIMOS LA CONSULTA Y ACCEDEMOS AL EDITOR DEL COMPONENTE TFDUPDATESQL.	294
9.17	SELECCIONAMOS LAS COLUMNAS PARA CADA CATEGORÍA.	294
9.18	AJUSTAMOS LAS SENTENCIAS SQL SI ES PRECISO.	295
9.19	SOLICITAMOS AL SERVIDOR LOS DATOS AGREGADOS Y AGRUPADOS.	297
9.20	CONFIGURAMOS EL PARÁMETRO A FACILITAR A LA CONSULTA.	300
9.21	EL PROGRAMA EJECUTANDO LA FUNCIÓN.	301
9.22	PROPIEDADES DEL COMPONENTE TFDSCRIPT.	304
9.23	EDICIÓN DEL *script* SQL.	305
9.24	OPCIONES PARA PROBAR EL GUIÓN EN LA FASE DE DISEÑO.	305
9.25	CUADRO DE DIÁLOGO CON LA EJECUCIÓN DEL SCRIPT.	308
10.1	PROPIEDADES QUE CONTROLAN LA GESTIÓN DE TRANSACCIONES.	316
10.2	NO ES POSIBLE OBTENER UN BLOQUEO SOBRE LA FILA.	322
10.3	PROPIEDADES DEL COMPONENTE TFDEVENTALERTER.	324
10.4	NUEVAS PROPIEDADES EN EL COMPONENTE TFDQUERY.	326
10.5	CONFIGURACIÓN DEL EVENTO A VIGILAR EN LA BASE DE DATOS.	328

10.6 FORMULARIO PARA CONTROLAR TRANSACCIONES Y EDITAR LAS CATEGORÍAS. 329

11.1 FORMULARIO PARA PROBAR EL MODO DESCONECTADO. 337

11.2 ELEMENTOS EN LA INTERFAZ DE USUARIO. 345

11.3 LA MODIFICACIÓN SIMULTÁNEA GENERA UN ERROR. 348

11.4 ENLAZAMOS EL MÓDULO DE DATOS CON EL NUEVO FORMULARIO. 352

11.5 ELEMENTOS DEL FORMULARIO DE RESOLUCIÓN DE CONFLICTOS. 353

11.6 LISTA DE ACCIONES EN EL TCOMBOBOX. 353

11.7 EL PROGRAMA INTENTADO SOLVENTAR UN CONFLICTO. 356

12.1 DISTRIBUCIÓN DEL CLIENTE INTERBASE CON LA APLICACIÓN. 362

12.2 CONFIGURACIÓN DEL CONTROLADOR FIREDAC PARA INTERBASE. 363

12.3 INFORMACIÓN SOBRE INTERBASE SERVER. 365

12.4 CONFIGURACIÓN DE LA CODIFICACIÓN A EMPLEAR. 368

12.5 OPCIONES DE MANTENIMIENTO. 369

12.6 COMPONENTES DE SERVICIO PARA INTERBASE. 369

13.1 ARQUITECTURA DE UNA SOLUCIÓN BASADA EN DATASNAP. 377

13.2 DOCUMENTO JSON OBTENIDO A PARTIR DEL TJSONOBJECT. 391

14.1 COMPONENTES PARA SERVIDORES DATASNAP. 396

14.2 OPCIÓN PARA INICIAR EL ASISTENTE DE SERVIDOR DATASNAP. 398

14.3 SELECCIÓN DEL TIPO DE CONTENEDOR. 399

14.4 COMPONENTES A INCLUIR EN EL SERVIDOR. 400

14.5 CONFIGURAMOS Y PROBAMOS EL PUERTO DE COMUNICACIÓN. 401

14.6 COMPONENTES EN EL MÓDULO CONTENEDOR. 402

14.7 EL CORTAFUEGOS DE WINDOWS NOTIFICA QUE EL SERVIDOR DEMANDA COMUNICACIÓN POR RED. 404

14.8 DEFINIMOS LA CONEXIÓN PARA ACCEDER AL SERVIDOR DATASNAP. 405

14.9 EXAMINAMOS EL SERVICIO DESDE EL EXPLORADOR DE DATOS. 406

14.10 COMPOSICIÓN DE LA INTERFAZ Y COMPONENTES DEL CONSUMIDOR. 407

14.11 LANZAMOS EL ASISTENTE PARA GENERAR EL MÓDULO CLIENTE. 409

14.12	ESPECIFICAMOS EL TIPO DE SERVIDOR DATASNAP AL QUE SE QUIERE ACCEDER.	410
14.13	FACILITAMOS LOS PARÁMETROS DE COMUNICACIÓN CON EL SERVIDOR.	411
14.14	AGREGAMOS AL FORMULARIO REFERENCIAS A LOS MÓDULOS CLIENTE.	412
14.15	EL CLIENTE CON LAS DOS VÍAS PARA CONSUMIR EL SERVICIO.	413
14.16	PODEMOS CONTROLAR EL ESTADO DEL SERVIDOR.	415
14.17	LA CLASE CON LOS MÉTODOS DE SERVICIO SERÁ DESCENDIENTE DE TDSSERVERMODULE.	416
14.18	COMPONENTES EN EL MÓDULO DE CLASES DEL SERVIDOR DATASNAP.	417
14.19	TOMAMOS EL PROCEDIMIENTO DEL EXPLORADOR DE DATOS.	419
14.20	INTRODUCIMOS LA CONSULTA QUE QUEREMOS EJECUTAR.	420
14.21	OBTENEMOS LOS DATOS DEL SERVIDOR EN EL DISEÑADOR.	421
15.1	SELECCIONAMOS EL TIPO DE APLICACIÓN CONTENEDORA DEL SERVIDOR.	427
15.2	ELEGIMOS LOS ELEMENTOS A INCLUIR EN EL PROYECTO.	428
15.3	COMPONENTES AGREGADOS POR EL ASISTENTE AL MÓDULO WEB.	430
15.4	CONFIGURACIÓN DE LAS EXTENSIONES DE ARCHIVOS ACEPTADAS.	431
15.5	SELECCIÓN DEL TIPO DE *proxy* A GENERAR.	432
15.6	EDICIÓN DE LAS ACCIONES ASOCIADAS AL MÓDULO WEB.	433
15.7	CONFIGURAMOS LAS PETICIONES QUE SE ATENDERÁN.	435
15.8	FORMULARIO DE CONTROL DEL SERVIDOR DATASNAP REST.	436
15.9	LLAMADA AL SERVICIO DESDE EL NAVEGADOR CON PETICIÓN GET.	438
15.10	LLAMADA DE TIPO POST CON EL FORMULARIO DE INVOCACIÓN.	439
15.11	EJECUTAMOS LA SOLICITUD DESDE EL DISEÑADOR DE DELPHI.	443
15.12	EXAMINAMOS LA RESPUESTA OBTENIDA DEL SERVICIO.	443
15.13	INTERFAZ DE USUARIO DEL CLIENTE REST DELPHI.	444
15.14	CONFIGURACIÓN DE LA PETICIÓN PARA CONSUMIR EL SERVICIO.	446
15.15	SE FACILITAN LOS PARÁMETROS QUE PRECISA EL SERVICIO.	447
15.16	CONFIGURAMOS EL COMPONENTE TPAGEPRODUCER.	456

15.17	Agregamos una nueva acción al módulo web para devolver la página.	457
15.18	El cliente REST jQuery Mobile en funcionamiento.	458
15.19	Componentes a incluir en el módulo de datos del servidor.	460
15.20	Documento JSON conteniendo el conjunto de datos.	463
15.21	El proyecto necesita un módulo REST de cliente para acceder al servidor.	464
15.22	Componentes en el módulo de datos.	465
15.23	El cliente puede modificar los datos facilitados por el servidor REST.	467
16.1	Ejecutables con la versión de desarrollo del servidor y consola EMS.	472
16.2	Componentes para acceder a los servicios de EMS.	473
16.3	Asistentes para desarrollo de paquetes EMS.	473
16.4	Primer paso del asistente de configuración de EMS.	475
16.5	El asistente confirma la configuración de EMS.	475
16.6	Consola de control del servidor EMS.	476
16.7	Programa de control de la consola de desarrollo.	477
16.8	Consola de desarrollo de EMS.	477
16.9	En el registro de eventos aparecen los recursos registrados.	478
16.10	Recuperamos la lista de grupos registrados.	479
16.11	Obtenemos detalles de un grupo concreto.	480
16.12	Creamos un nuevo grupo con la solicitud de tipo `POST`.	480
16.13	Añadimos un usuario nuevo al grupo creado antes.	481
16.14	Iniciamos sesión en el servidor EMS.	482
16.15	Facilitamos al servicio el *token* obtenido tras iniciar sesión.	483
16.16	En el registro se aprecia el inicio de sesión.	483
16.17	Campos asociados al registro de instalaciones.	484
16.18	Métodos para operar sobre la API de grupos del servidor EMS.	485
16.19	El cliente EMS en funcionamiento.	489

16.20	Último paso del asistente para paquetes EMS.	491
16.21	Probamos el servidor EMS de números aleatorios.	494
A.1	Elementos fundamentales del IDE de Delphi.	501
A.2	Herramientas de personalización de los paneles.	503
A.3	Guardamos la configuración actual del IDE.	504
A.4	Búsqueda con IDE Insight.	505
A.5	Tipos de proyecto disponibles.	507
A.6	Búsqueda de la opción con IDE Insight.	507
A.7	Proyectos y plantillas en la Paleta de herramientas.	508
A.8	Plantillas de proyecto multi-dispositivo.	509
A.9	El Gestor de proyectos.	510
A.10	Agregamos un nuevo proyecto al grupo actual.	511
A.11	Lista con las vistas de diseño disponibles.	512
A.12	Podemos rotar la vista y desactivar la máscara de dispositivo.	513
A.13	Importación de los módulos de las vistas.	515
A.14	Visualización preliminar para cada plataforma.	516
A.15	Agregamos al diseñador una nueva vista.	517
A.16	Componentes en la Paleta de herramientas.	518
A.17	Relación entre los componentes alojados en el contenedor.	519
A.18	Vista parcial del Inspector de objetos.	522
A.19	Editor de listas de cadenas de caracteres.	523
A.20	Editor específico para una propiedad de tipo TBrush.	524
A.21	Lista de eventos de un componente TFDConnection.	525
A.22	Lista de objetos accesibles en el contexto actual.	527
A.23	Lista de miembros de un objeto.	528
A.24	Lista de parámetros que necesita el método.	528
A.25	Valores válidos para una propiedad enumerada.	529
A.26	Lista de plataformas objetivo y su configuración.	530

A.27	SELECCIONAMOS LA PLATAFORMA ACTIVA.	531
A.28	AGREGAMOS UNA PLATAFORMA OBJETIVO AL PROYECTO.	532
A.29	SELECCIONAMOS LA CONFIGURACIÓN DE GENERACIÓN DEL PROYECTO.	533
A.30	ACTIVAMOS UN PUNTO DE INTERRUPCIÓN.	535
A.31	CONFIGURACIÓN DE PUNTO DE INTERRUPCIÓN.	536
A.32	COMANDOS DE EJECUCIÓN PASO A PASO.	537
A.33	EL VALOR DE LA VARIABLE APARECE EN UNA VENTANA EMERGENTE.	538
A.34	CUADRO DE DIÁLOGO PARA EVALUACIÓN DE EXPRESIONES.	539
A.35	VENTANA CON VARIABLES BAJO SUPERVISIÓN.	540
A.36	INFORMACIÓN SOBRE LAS LLAMADAS HASTA EL PUNTO ACTUAL.	541
B.1	FORMULARIOS EN EL PROYECTO DE EJEMPLO FORMS.	550
B.2	SEGUIMIENTO AUTOMÁTICO DE LOS COMENTARIOS TIPO TODO.	553
B.3	LOS ELEMENTOS DEL TIPO ENUMERADO TIENEN ASOCIADO UN VALOR.	561
B.4	EL VALOR ESTÁ FUERA DE LOS LÍMITES DEL SUBRANGO.	565
B.5	LA OPCIÓN **RANGE CHECKING** ESTÁ DESHABILITADA POR DEFECTO.	566
B.6	CODE INSIGHT MUESTRA LOS MIEMBROS DEL REGISTRO.	572
B.7	AGREGAMOS UN MÓDULO DE CÓDIGO AL PROYECTO.	598
B.8	PLANTILLA DE CÓDIGO PARA BUCLE FOR.	626
B.9	LISTA CON LAS PLANTILLAS DE CÓDIGO PREDEFINIDAS.	627
B.10	SELECCIÓN DE UN PERFIL PARA DAR FORMATO AL CÓDIGO.	630
B.11	PODEMOS COMPARAR VERSIONES Y RESTAURARLAS.	631
C.1	PÁGINA DESDE LA QUE DESCARGAREMOS GIT.	635
C.2	CONFIGURACIÓN DE LA LÍNEA DE COMANDOS DE GIT.	636
C.3	ACCESO A LA GUI Y LÍNEA DE COMANDOS DE GIT.	637
C.4	CONFIGURACIÓN INICIAL DE GIT.	638
C.5	CONFIGURACIÓN DE GIT EN EL ENTORNO DE DELPHI.	639
C.6	CLONACIÓN DE UN REPOSITORIO REMOTO.	641
C.7	PROCESO DE CLONACIÓN DEL REPOSITORIO REMOTO.	642

C.8	Obtener dirección de clonación desde Github.	642
C.9	Menú Git agregado al menú contextual de los módulos.	645
C.10	Configuramos la nueva opción para invocar a `git init`.	646
C.11	Usamos la opción para inicializar el nuevo repositorio local.	647
C.12	Opciones de Git asociadas al proyecto.	648
C.13	Lista de módulos con cambios y agregados al proyecto.	649
C.14	Menú contextual asociado a cada módulo en la ventana Commit.	649
C.15	Revisión de los cambios hechos a un módulo.	650
C.16	Actualizamos nuestro repositorio local.	652
D.1	Descarga del instalador de BDE para Delphi XE8.	659
D.2	Instalación de BDE en un equipo con Delphi XE8.	659
D.3	Localizamos el paquete agregado por el instalador.	660
D.4	Activamos el paquete de componentes BDE.	661
D.5	Componentes BDE en la Paleta de herramientas.	661
D.6	Herramienta para importar alias.	666

SOBRE EL AUTOR

Francisco Charte Ojeda, Ingeniero informático por la Universidad de Jaén y Doctor en Tecnologías de la información y la comunicación por la Universidad de Granada, es autor de más de un centenar de libros sobre sistemas operativos, lenguajes de programación y tecnología, publicados en los últimos 30 años por distintas editoriales. Su experiencia con Delphi se inicia antes de que se lanzase la primera versión, en 1995. Desde entonces ha escrito una veintena de libros sobre las distintas versiones de esta herramienta, incluyendo títulos centrados en el entorno y lenguaje, otros en el desarrollo de aplicaciones móviles y algunos más dedicados al trabajo con bases de datos.

AGRADECIMIENTOS

A pesar de que son muchos los años escribiendo, más de treinta ya, y de que este es mi centésimo vigésimo segundo libro, la experiencia sigue siendo tan maravillosa como cuando terminé de escribir el primero. La finalización de este conlleva una enorme satisfacción, de la que me gustaría hacer partícipes a todos aquellos que lo han hecho posible. Mi más sincero agradecimiento a José Luis y los compañeros de Danysoft por confiarme este proyecto, a mis amigos por su comprensión durante estos meses y, sobre todo, a mi familia, por su eterno, extraordinario e incondicional apoyo.

Capítulo 1

INTRODUCCIÓN

Delphi siempre se ha caracterizado, desde su primera versión, por ofrecer al desarrollador mecanismos que simplifican el trabajo con distintas bases de datos. Actualmente, 20 años después del lanzamiento de su primera versión, el acceso a bases de datos sigue siendo uno de los puntos fuertes de Delphi, así como la razón para que muchos profesionales lo elijan como entorno de trabajo.

Si algo ha cambiado en estas dos últimas décadas es la forma en que usamos las aplicaciones. Hace dos décadas el dispositivo por excelencia, casi el único disponible, era el ordenador personal. La información sobre la que operaban los programas se encontraba almacenada en el mismo equipo, en una base de

datos local, o bien en un servidor de bases de datos departamental, generalmente conectado a la misma red local. El escenario actual es mucho más heterogéneo, comenzando por los tipos de dispositivos usados para ejecutar aplicaciones: ordenadores, tabletas, teléfonos móviles e incluso dispositivos de tipo *wearable* como los nuevos relojes y pulseras inteligentes. Las aplicaciones utilizadas son nativas en unos casos y aplicaciones web en otros, de forma que la interfaz de usuario se abre en un navegador web. La información sobre la que operan estos programas puede estar almacenada localmente en el mismo dispositivo, pero también encontrarse en un servidor remoto, en la otra esquina del mundo, o estar distribuida.

El conjunto de bibliotecas de componentes para acceso a datos incorporadas por Delphi ha ido evolucionando a lo largo del tiempo, dando cobertura a todas esas nuevas necesidades. El objetivo de este libro es mostrar al lector cómo puede usar Delphi en los diferentes contextos esbozados en las tres siguientes secciones de esta introducción.

1.1 Aplicaciones móviles

El número de dispositivos móviles en uso ya supera al número total de personas que habitan el planeta. Es habitual que aparte de nuestro ordenador personal también contemos con una tableta, obviamente un teléfono móvil y quizá también con un reloj inteligente. Todos estos *gadgets* tienen, en mayor o menor medida, capacidad para ejecutar aplicaciones, programas que, por regla general, necesitarán datos para realizar su trabajo.

> **NOTA**
>
> *Con Delphi podemos crear aplicaciones móviles nativas para iOS y Android, así como aplicaciones web accesibles desde prácticamente cualquier dispositivo, móvil o no.*

La información usada por una aplicación móvil puede alojarse en el propio dispositivo en que se ejecuta. Existen versiones específicas de gestores de bases de datos relacionales (RDBMS en adelante) para los sistemas operativos móviles

más difundidos. Un ejemplo de ello es **InterBase ToGo**. En este escenario los programas precisan un mecanismo de acceso a datos local, ya que el RDBMS se ejecuta en la misma máquina, no siendo necesario el uso de conexiones hacia el exterior.

Un programa que se ejecuta en un dispositivo móvil también puede conectar con un RDBMS remoto, operando en este caso bajo la tradicional arquitectura cliente/servidor. Para ello el dispositivo debería contar con una conexión a la red donde se ejecuta el RDBMS, ya sea directa o a través de Internet.

1.2 Aplicaciones Windows/OS X

Las aplicaciones de escritorio, diseñadas para aprovechar la potencia de los ordenadores que ejecutan Windows y OS X, han sido tradicionalmente las que más necesidades de mecanismos de acceso a datos han presentado. Actualmente algunas tabletas y teléfonos son prácticamente tan potentes como estos ordenadores, pero existe una gran diferencia entre ambos tipos de dispositivos: la capacidad de almacenamiento.

Un ordenador, ya sea portátil o de escritorio, suele contar con un uno o más discos para almacenamiento de datos, con una capacidad total que multiplica por varias veces las de los móviles y tabletas. Estos últimos raramente superan los 32-64 GB[1] de almacenamiento, mientras que la mayoría de ordenadores modernos cuentan ya con 1 TB o más. Por ello resulta bastante corriente que el usuario almacene en su propia máquina gran parte de la información con que ha de trabajar, empleando aplicaciones que operan sobre bases de datos locales.

Lo anterior es especialmente cierto cuando la actividad del usuario no está vinculada a alguna empresa u organización, sino que se desempeña de manera individual. En las empresas, sobre todo a partir de cierto tamaño, los ordenadores suelen estar conectados a un servidor de datos departamental o central, dependiendo de la estructura de la organización. Son situaciones en las que se recurre principalmente a la arquitectura cliente/servidor, con RDBMS como Oracle o

[1]Estas configuraciones son las usuales en 2015, pero probablemente dicha capacidad se incremente de manera significativa en el futuro.

SQL Server que, dependiendo de la configuración hardware, están preparados para atender a miles de usuarios de forma simultánea.

1.3 Aplicaciones web

En las dos secciones previas se asume que las aplicaciones son nativas[2], por lo que tienen acceso a los servicios necesarios para conectar con el RDBMS, ya sea local o remoto. Si el programa en cuestión es una aplicación web, accesible desde cualquier tipo de dispositivo, lo que ejecuta en el equipo del usuario es únicamente la interfaz, existiendo otra parte que opera en un servidor y que, a su vez, conectará con el RDBMS. Este es posiblemente el escenario más habitual en la actualidad: el de una aplicación distribuida.

> **NOTA**
>
> *Las aplicaciones web también pueden almacenar información localmente, ya sea mediante cookies o usando desde JavaScript el servicio* `localStorage` *de HTML5.*

Las aplicaciones distribuidas se estructuran en varias capas, de ahí que en ocasiones también se use la denominación *aplicación multi-capa*. La arquitectura más habitual es la que utiliza tres capas:

- La **interfaz de usuario** es la capa que se ejecuta en el dispositivo del usuario, ya sea este un móvil, una tableta o un ordenador.
- En el servidor web se ejecuta la capa con la **lógica de procesamiento**, encargada de responder a las solicitudes de la capa de interfaz. Para ello se puede usar el mecanismo tradicional, sirviendo páginas, o bien recurrir el esquema REST (*Representational State Transfer*) habitual de los servicios web.

[2]En algunos casos puede ser preciso un intérprete o *runtime*, como ocurre con las aplicaciones Java que precisan de la máquina virtual o JVM, pero asumiremos que estas también son aplicaciones nativas.

- La capa anterior será la que, cuando se precise, conecte con el RDBMS que se ejecuta en la tercera capa, la **capa de datos**. Aunque habitualmente estas dos últimas capas se ejecutarán en servidores conectados entre sí localmente, también cabe la posibilidad de que el servidor de datos sea un servicio remoto respecto al servidor web.

Si bien esta arquitectura es, como se ha dicho antes, la más usual, nada impide la existencia de capas adicionales. Además es una arquitectura no exclusiva de las aplicaciones web, también puede utilizarse con aplicaciones móviles o de escritorio. La principal diferencia estriba en que la interfaz de usuario será una GUI (*Graphics User Interface*) nativa, en lugar de un documento HTML/CSS que es necesario abrir en un navegador web.

1.4 Servicios de acceso a bases de datos y Delphi

La incorporación de mecanismos propios de acceso a bases de datos ha sido siempre uno de los puntos fuertes de Delphi desde su lanzamiento, en 1995, convirtiéndose así en la herramienta ideal para el desarrollo de las clásicas *aplicaciones de gestión*, ocupando el espacio dejado por productos míticos como dBase, FoxPro o Clipper.

Con el paso del tiempo las necesidades de los programadores han ido evolucionando, a medida que las aplicaciones nativas de escritorio fueron dejando paso a las aplicaciones web, en primer término, y a las móviles, en segundo. Los servicios de acceso a datos ofrecidos por Delphi también se fueron adecuando a esas necesidades, haciendo posible el diseño de soluciones distribuidas y también embebidas.

Esta sección es un viaje rápido por la historia de los servicios de acceso a bases de datos incorporados por Delphi a lo largo de los años, desde el hoy obsoleto BDE hasta el actual FireDAC.

El objetivo que se persigue con este recorrido es obtener una visión general que nos permita comprender cómo se ha llegado al punto actual, así como la razón de que FireDAC sea el mecanismo que deberíamos utilizar preferente en

el desarrollo de nuestros nuevos proyectos, si bien todas las demás opciones permanecen también abiertas.

> **NOTA**
>
> *No es necesario conocer absolutamente nada sobre BDE, IBX y los demás motores de acceso a datos para seguir este libro y comenzar a trabajar con FireDAC en las actuales versiones de Delphi.*

1.4.1 BDE

BDE (*Borland Database Engine*) es la denominación del primer mecanismo de acceso a datos ofrecido por Delphi. BDE era un producto basado en bibliotecas de acceso a datos usadas previamente por Borland en Paradox[3] y en Turbo Pascal[4], cuya interfaz de programación era conocida como **IDAPI** (*Integrated Database Application Program Interface*).

Mediante BDE las aplicaciones podían acceder directamente a bases de datos dBase, Paradox, Access y FoxPro, gestores de datos locales todos ellos. Si se agregaba el módulo *SQL Links* también se podía acceder a RDBMS como DB2, Informix, InterBase, Sybase y Oracle en una configuración cliente/servidor. Asimismo se ofrecía un controlador que actuaba como puente con ODBC (*Open Database Connectivity*), abriendo así la puerta a usar cualquier base de datos accesible mediante esta vía. En la Figura 1.1 se ha representado la estructura típica de las soluciones basadas en BDE.

Con BDE Delphi introdujo una arquitectura de componentes que terminó influenciando el desarrollo de otras soluciones de acceso a datos (véanse los apartados siguientes). La base de datos estaba representada por un componente `TDatabase`, en el que se alojaba la información necesaria para conectar con

[3] Paradox era una aplicación de gestión de bases de datos local, la categoría a la que pertenece Microsoft Access, y en su tiempo competía con el entonces rey del mercado que era dBase. Finalmente Borland terminó adquiriendo en 1991 la empresa Asthon Tate, desarrolladora de dBase.

[4] El predecesor de Delphi contaba con una biblioteca complementaria de acceso a datos conocida como *Database Toolbox*.

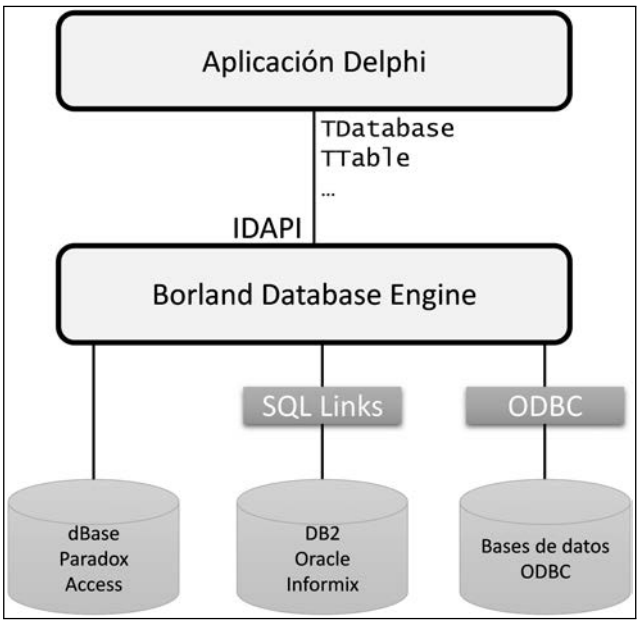


Figura 1.1 ARQUITECTURA DE UNA SOLUCIÓN BASADA EN BDE

ella. Mediante los componentes `TTable` y `TQuery` se operaba sobre tablas de datos y con consultas SQL. Un componente `TDataSource`, actuando como intermediario con los anteriores, permitía conectar los datos con los controles que formaban la interfaz de usuario.

NOTA

BDE no solo incluía los componentes necesarios para comunicarse con las bases de datos enumeradas, sino que también incorporaba el motor de dBase y Paradox. Esto permitía a las aplicaciones Delphi trabajar con esos tipos de bases de datos sin necesidad de instalar dichos productos. A cambio la instalación de BDE incluye elementos que pueden no ser necesarios para nuestros proyectos.

BDE es una solución lanzada hace ahora más de dos décadas y cuyo desarrollo se detuvo hace tiempo. Los mayores inconvenientes de este mecanismo de acceso a datos son:

- **Depencia del sistema operativo**: BDE y sus controladores son una solución específica **Win32**. Esto significa que no podremos usarlo en aplicaciones de 64 bits desarrolladas para Windows, ni tampoco en otros sistemas como OS X, Android o iOS. Optar por BDE, por tanto, implicaría perder una de las grandes ventajas que nos ofrecen las últimas versiones de Delphi: el desarrollo de soluciones multi-dispositivo.
- **Despliegue e instalación complejos**: La instalación de BDE en los equipos donde va a desplegarse una aplicación Delphi es, aparte de relativamente pesada por la inclusión de los motores de dBase y Paradox, compleja[5], ya que se precisa escribir datos de configuración en el registro de Windows, para lo cual es necesario contar con privilegios de administrador. Si hay más de una aplicación usando BDE en el sistema pueden aparecer conflictos entre ellas a raíz de esos parámetros de configuración.
- **Sin Unicode**: BDE no contempla el uso de Unicode ni lo hará nunca, lo cual complica su uso con las versiones recientes de Delphi[6], aparte de limitar nuestras aplicaciones al no poder utilizar alfabetos que usan caracteres no existentes en las codificaciones tradicionales.
- **Sin soporte**: El desarrollo de BDE se detuvo hace mucho tiempo. Delphi 6, lanzado en el año 2000, incorporó como novedad **dbExpress**, el mecanismo de acceso a datos llamado a sustituir a BDE. En el año 2002 SQL Links fue declarado como obsoleto (*deprecated*). BDE es, por tanto, una solución anticuada y que no evolucionará más, para la que no se resolverán fallos y para la que el fabricante ya no da soporte.

Si las razones anteriores no son suficientes para decidirnos a abandonar BDE, debemos tener en cuenta que a partir de la versión XE7 BDE ya no se distribuye como parte del producto No obstante, puede descargarse e instalarse por separado (véase el Apéndice D), una opción que nos permitiría hacer la transición hacia alguna de las demás opciones discutidas a continuación.

[5]Antes de que una aplicación BDE pueda funcionar es preciso configurar los alias de bases de datos y configurar los controladores que vayan a utilizarse.

[6]El tipo `String` de Delphi usa por defecto Unicode desde Delphi 2009.

1.4.2 IBX

Además de dBase, la adquisición de Asthon Tate por parte de Borland también incorporó al abanico de productos de esta última empresa un RDBMS llamado **InterBase**[7]. Aunque Borland terminó deshaciéndose de sus otros productos de bases de datos, Paradox por ejemplo fue vendido a Novell, continuó desarrollando InterBase como el RDBMS propio de la empresa, enfoque que también ha mantenido Embarcadero. Actualmente contamos con versiones de InterBase tanto para servidores de datos como para operar bases de datos embebidas, incluso en dispositivos móviles (el antes mencionado InterBase ToGo).

Dado que el fabricante de Delphi contaba con un RDBMS propio, era lógico que la herramienta de desarrollo incorporase componentes capaces de aprovechar las características específicas de dicho RDBMS. Esa biblioteca de componentes a medida es **IBX** (*InterBase eXpress*), lanzada en versión beta junto con Delphi 5 y que ha formado parte de todas las versiones de Delphi posteriores.

Los componentes IBX están diseñados para comunicarse directamente con el software cliente de InterBase (véase la Figura 1.2), lo cual hace innecesarios los controladores intermedios existentes en BDE. En consecuencia el acceso a los datos pasa por menos capas, lo cual redunda en un mayor rendimiento. Una ventaja adicional es que las aplicaciones Delphi pueden aprovechar las características específicas que ofrece InterBase, incluso existen componente IBX para tareas de administración, prescindiendo así de herramientas externas.

> **NOTA**
>
> *Los componentes IBX se agrupan en dos categorías: componentes de acceso a datos, equivalentes a los que ofrece BDE, y componentes de administración. Estos últimos permiten a las aplicaciones obtener información estadística del funcionamiento del servidor, efectuar copias de seguridad de las bases de datos, etc., usando para ello las funciones ofrecidas por la API de InterBase.*

[7] Asthon Tate adquirió en 1988 la tecnología de InterBase de Interbase Software Corporation, una empresa fundada en 1984 centrada en la creación de software de bases de datos relacionales para estaciones de trabajo Sun y sistemas DEC VAX.

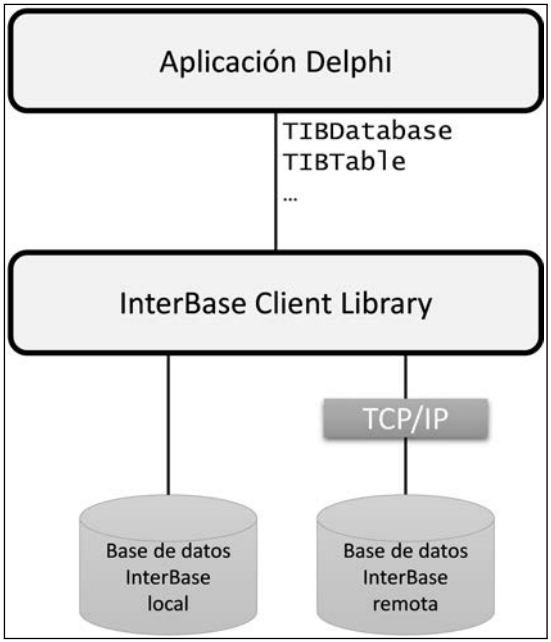


Figura 1.2 ARQUITECTURA DE UNA SOLUCIÓN BASADA EN IBX.

La principal, y casi la única, desventaja de IBX es que se trata de una solución a medida de un RDBMS. Desarrollar una aplicación Delphi usando IBX dificultará una futura transición a otro RDBMS, ya que nos veríamos forzados a usar otro mecanismo de acceso a datos. Esto supondría cambiar todos los componentes utilizados en el proyecto por otros distintos.

1.4.3 dbGo

En la segunda mitad de los años 90 Microsoft desarrolló para Windows sus propios servicios de acceso a datos, apoyándose para ello en la infraestructura COM (*Component Object Model*), su modelo de componentes software. Es en este contexto donde aparecen los componentes ADO (*ActiveX Data Objects*) y los controladores OLE DB, ofreciendo una solución de acceso a bases de datos de

tipo genérico, independientes tanto de la base de datos como del lenguaje de programación utilizado. La interfaz para las aplicaciones era uniforme, sólo se precisaba cambiar de controlador OLE DB para operar sobre un RDBMS u otro.

A fin de aprovechar estos nuevos servicios, Delphi incorporó en su versión 5 un nuevo conjunto de componentes denominados **ADOExpress**. La biblioteca cambió su nombre por **dbGo** en Delphi 6, denominación que ha mantenido hasta la actualidad.

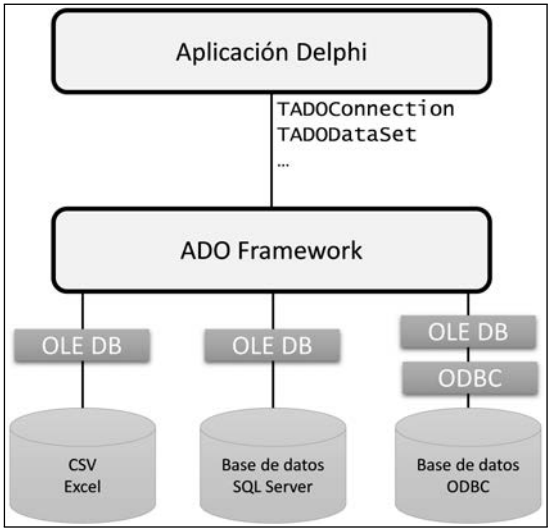


Figura 1.3 ARQUITECTURA DE UNA SOLUCIÓN BASADA EN DBGO.

Los componentes dbGo actúan a modo de envoltorios (*wrappers*) en torno a los servicios ofrecidos por ADO, facilitando la comunicación con los distintos proveedores OLE DB disponibles. Un proveedor OLE DB es como un controlador BDE, actuando como intermediario entre la aplicación y el software cliente de la base de datos con la que vaya a trabajarse. Hay disponibles proveedores OLE DB para diversos RDBMS, entre ellos Microsoft SQL Server, así como un puente OLE DB-ODBC que facilita el acceso a cualquier origen de datos para el que exista un controlador ODBC.

El inconveniente de dbGo es que se trata de una solución específica de un fabricante, válida únicamente para aplicaciones que se ejecutarán en equipos con Windows. Cualquier proyecto con objetivos multiplataforma, por tanto, descarta ya de entrada esta solución. Sin embargo para aplicaciones que únicamente van a utilizarse en Windows dbGo puede ser una buena alternativa, dado que facilita el trabajo con fuentes de datos muy habituales en dicho sistema operativo como son las hojas de cálculo Excel o las bases de datos Access.

1.4.4 dbExpress

El lanzamiento de Delphi 6 vino acompañado de un nuevo producto que había sido esperado durante mucho tiempo por los usuarios de Delphi: una versión para GNU/Linux de su entorno de desarrollo preferido, con su correspondiente compilador, biblioteca de componentes y otras utilidades. Ese nuevo producto se denominó **Kylix** y conllevó algunas novedades más. Quizá las más destacables fueron la biblioteca de componentes *cross-platform* **CLX** y el nuevo mecanismo de acceso a bases de datos **dbExpress**.

La VCL es una biblioteca de componentes nacida en Windows y con estrechos vínculos con el sistema operativo de Microsoft, razón por la que hubo que desarrollar una nueva, la mencionada CLX, pensada para ser multiplataforma desde un principio. La CLX estaba llamada a sustituir a la VCL, pero el limitado éxito de Kylix, del que solo existieron dos versiones más, provocó que tanto el producto como la biblioteca de componentes fuesen abandonadas.

> **NOTA**
>
> *La última versión de Kylix, la versión 3, fue presentada en 2002 y sus características eran similares a las de Delphi 7, en cuanto a compiladores (se incluía Delphi y C++), entorno de desarrollo, biblioteca CLX y software de acceso a datos dbExpress.*

Análogamente, dbExpress fue diseñada como una solución de acceso a datos multiplataforma, sin vínculos con Windows a diferencia de BDE, y su objetivo era sustituir a esta última. En contraste con la suerte que corrió la CLX, dbExpress siguió evolucionándose en versiones posteriores de Delphi y sigue siendo

una de las opciones existentes actualmente a la hora de desarrollar aplicaciones con acceso a datos. La principal ventaja de dbExpress es que se trata de una solución escrita totalmente en el propio lenguaje Delphi, por lo que los controladores necesarios para acceder a las bases de datos pueden integrarse directamente en la aplicación.

Aunque en la actualidad no existe una versión de Delphi para GNU/Linux, sí que es posible desarrollar aplicaciones para OS X. Estas aplicaciones multiplataforma no pueden usar BDE o dbGo, pero sí dbExpress ya que esta es una solución que puede compilarse para distintos sistemas.

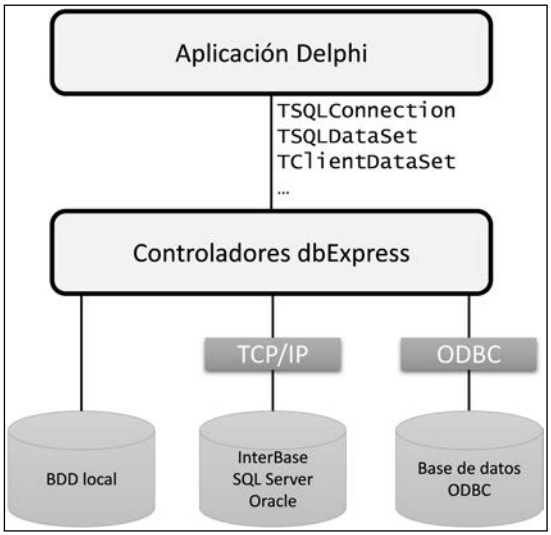


Figura 1.4 ARQUITECTURA DE UNA SOLUCIÓN BASADA EN DBEXPRESS.

Los controladores dbExpress están diseñados en base a un mínimo común denominador de los distintos RDBMS, por lo que no aprovechan las características específicas de cada uno de ellos. Además fueron desarrollados con el rendimiento como principal objetivo, sacrificando en algunos casos funciones más avanzadas a cambio de ser más ágiles en el tratamiento de los datos.

1.4.5 DataSnap

Los distintos *frameworks* de acceso a bases de datos mencionados en los apartados previos facilitan el acceso a datos locales o bien alojados en un servidor, según la tradicional arquitectura cliente/servidor. La necesidad de centralizar el acceso a los datos en un servidor independiente del que ejecuta el RBDMS, a fin de optimizar el uso de recursos, no es nueva y dio origen a la arquitectura distribuida en varias capas o *multi-tier*.

Ya en Delphi 3 se ofrecía una solución llamada a satisfacer dicha necesidad: MIDAS (*Multi-tier Distributed Application Service Suite*). Esta cambió su nombre por DataSnap en Delphi 6, denominación que mantiene a día de hoy. Originalmente DataSnap ofrecía fundamentalmente los servicios necesarios para operar de manera remota sobre conjuntos de datos, transportando la información necesaria en ambos sentidos: desde el servidor de aplicaciones al cliente y viceversa. Posteriormente se agregó la funcionalidad de poder ejecutar de forma remota funciones (servicios) definidos en el servidor de aplicaciones.

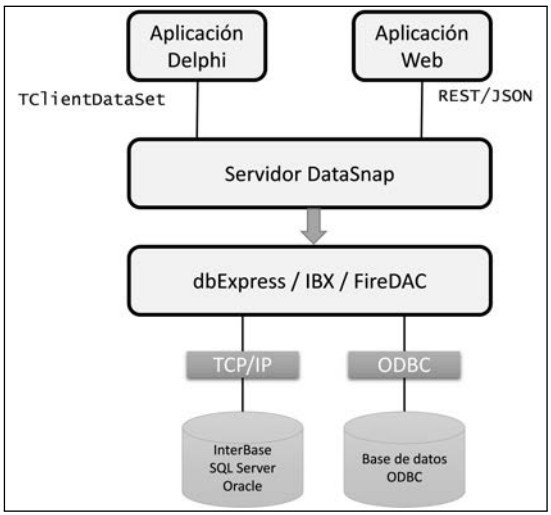


Figura 1.5 ARQUITECTURA DE UNA SOLUCIÓN BASADA EN DATASNAP

Originalmente MIDAS se diseñó para funcionar sobre DCOM (*Distributed Componente Object Model*), el modelo de componentes distribuido de Microsoft, por lo que dependía de la configuración DCOM del sistema para su correcto funcionamiento. Desde Delphi 2009 la implementación de DataSnap está basada en los componentes Indy TCP, por lo que es posible comunicarse por cualquier tipo de red basada en TCP/IP, sin dependencias respecto a DCOM. En Delphi 2010 se añadió a DataSnap soporte para solicitudes de tipo REST (*Representational State Transfer*) que en la actualidad, junto con JSON (*JavaScript Object Notation*), es el estándar para el desarrollo de servicios web.

El modelo distribuido ofrecido por DataSnap también ha evolucionado con el tiempo, adaptándose a la nuevas necesidades del mundo móvil en que nos desenvolvemos actualmente. Gracias a los *DataSnap Mobile Connectors* es posible acceder a datos alojados en un servidor desde aplicaciones móviles. A esto hay que sumar la posibilidad de usar REST desde cualquier tipo de cliente, esté o no desarrollado con Delphi.

1.4.6 FireDAC

A pesar de ser la opción de incorporación más reciente en Delphi, en la versión XE4 se podía adquirir FireDAC por separado y a partir de la versión XE5 se incluyó como parte de la edición *Professional* y superiores, lo cierto es que FireDAC no es un desarrollo completamente nuevo. Anteriormente conocido como AnyDAC[8], esta biblioteca de componentes y controladores fue adquirida a otra empresa, heredando más de una década de experiencia en su desarrollo.

FireDAC es una solución de acceso a datos denominada como *universal*. Este apelativo se debe a que está disponible para todos los sistemas operativos con los que podemos crear aplicaciones con Delphi: Windows, OS X, iOS y Android, y facilita la conexión prácticamente con cualquier origen de datos: bases de datos locales y embebidas, múltiples RDBMS y cualquier otra fuente para la que exista un controlador ODBC o dbExpress.

Una de las grandes ventajas de FireDAC es que ofrece a los desarrolladores un modelo unificado de acceso a los datos, sin que importe el sistema o dónde esté alojada la información, pero al tiempo también facilita la explotación de

[8]La biblioteca de componentes AnyDac, diseñada para Delphi, C++ Builder y FreePascal, fue adquirida por Embarcadero a la empresa DA-SOFT en febrero de 2013.

características específicas de cada RDBMS. Además FireDAC cuenta con componentes para mantener bases de datos en memoria, incorporando su propio motor SQL para trabajar sobre ellas.

Entre otros productos, una aplicación Delphi que use FireDAC podrá acceder a bases de datos de escritorio Access, bases de datos embebidas SQLite e InterBase ToGo y RDBMS como SQL Server, Oracle, DB2, MySQL, InterBase y Firebird.

> **NOTA**
>
> *Según la edición de Delphi con que contemos el conjunto de controladores FireDAC incluido variará, algo que hemos de tener en cuenta a la hora de planificar el trabajo con un cierto RDBMS.*

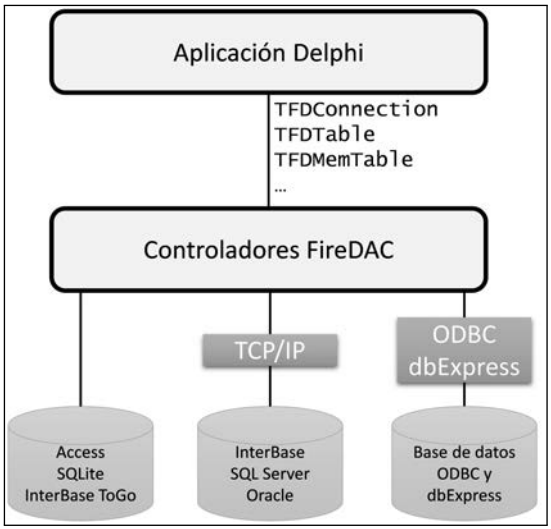


Figura 1.6 ARQUITECTURA DE UNA SOLUCIÓN BASADA EN FIREDAC

Por su enfoque multiplataforma, su rendimiento y las características avanzadas que ofrece, FireDAC es el mecanismo de acceso a datos preferente a la

hora de desarrollar nuevos proyectos con Delphi. Dependiendo de su complejidad, también puede merecer la pena la conversión de proyectos ya existentes, basados en BDE o en dbExpress, para aprovechar las ventajas de FireDAC. Debemos tener en cuenta que BDE es una tecnología totalmente obsoleta, como se apuntaba anteriormente, y que dbExpress, aunque probablemente siga aún mucho tiempo presente en Delphi, es una biblioteca cuyo desarrollo se ha estancado en favor de la evolución de FireDAC.

1.5 Sobre este libro

El objetivo de este libro es describir los componentes, las herramientas y los procedimientos a seguir para operar con bases de datos al desarrollar aplicaciones Delphi en distintos escenarios. Nuestra pretensión es ofrecer al lector la información que necesitará en cada caso concreto, explicando detalladamente cuál ser ía la configuración a usar y acompañando dichas explicaciones con ejercicios demostrativos que las pongan en práctica.

Como la práctica totalidad de libros sobre programación, este está pensado para ser leído prácticamente delante del ordenador, a fin de que se pueda ir experimentando cada paso personalmente. Para facilitar este seguimiento los proyectos propuestos como ejercicios están a disposición del lector en `www.danysoft.com/libros/DelphiBDD-master.zip` y en un repositorio de GitHub, en `github.com/fcharte/DelphiBDD`. Te recomendamos que comiences por obtener dichos proyectos para tenerlos al alcance del teclado a medida que avances.

NOTA

En el apéndice C se describe la instalación y uso de Git, así como la clonación de repositorios GitHub desde Delphi. No obstante, también puede descargar directamente el paquete con los ejemplos, sin recurrir a Git en `www.danysoft.com/libros/DelphiBDD-master.zip`

1.5.1 Estructura

A lo largo del libro se abordarán tres contextos de trabajo diferentes, correspondiéndose estos con las tres grandes partes en que está dividido el libro. A continuación se ofrece una descripción breve del contenido de cada una de ellas:

- **Acceso a datos locales**: Los capítulos de la primera parte tratarán el acceso a orígenes de datos locales, alojados en el mismo dispositivo en que se ejecuta la aplicación. Por una parte trabajaremos en proyectos dirigidos a ordenadores con Windows u OS X y, por otra, a aplicaciones para dispositivos móviles usando bases de datos embebidas. Siendo esta la configuracion más simple, ya que datos y programa están en la misma máquina, los capítulos de esta parte también explicarán cómo usar algunas herramientas y componentes fundamentales y de uso general.

- **Aplicaciones cliente/servidor**: En la segunda parte del libro nos ocuparemos de la arquitectura de acceso a datos más popular, aquella en la que la aplicación, ejecutándose en un dispositivo conectado a una red, se comunica con un servidor de datos para operar sobre la información. Además de los procedimientos de trabajo comunes a todos los RDBMS, que facilitan un desarrollo unificado, también se explicará cómo acceder a características específicas de productos concretos, usando para ellos componentes adecuados.

- **Aplicaciones distribuidas**: La tercera parte del libro se concentrará en las técnicas de acceso a datos en entornos distribuidos, con clientes que pueden ser aplicaciones Delphi en un ordenador pero también aplicaciones web en un navegador o programas para dispositivos móviles, un servidor de aplicaciones atendiendo peticiones de dichos clientes y un servidor de datos conectado al anterior. En los capítulos de esta parte conoceremos las distintas opciones a la hora de transportar y gestionar datos desde un extremo a otro.

Al final del libro, tras los capítulos de estas tres partes, encontramos cuatro apendices cuya finalidad es ayudar a los desarrolladores que no estén familiarizados con el lenguaje Delphi, con su entorno y con el uso de Git como sistema de control de versiones de código fuente. Nuestro objetivo con estos apéndices es facilitar al lector la información que pueda necesitar para comenzar a crear

aplicaciones Delphi con acceso a bases de datos, pero obviamente es imposible resumir en los mismos libros completos centrados en los aspectos más genéricos de esta herramienta.

1.5.2 Notación

En el texto de este libro se utilizarán distintas notaciones a fin de distinguir mejor ciertos tipos de elementos. En el texto propiamente dicho, aparte del tipo de letra normal, podemos encontrar también los siguientes estilos:

- **Destacado**: El texto en **negrita** se utiliza puntualmente para destacar algo importante o la primera vez que se introduce un concepto al que es necesario prestar atención.
- *Términos*: La letra en estilo *cursiva* se emplea para distinguir términos anglosajones o bien para introducir el significado de acrónimos en dicho idioma, habitualmente entre paréntesis siguiendo al acrónimo.
- `Código`: Este estilo diferencia en el texto los `nombres` de componentes, variables, sentencias de Delphi y, en general, cualquier elemento que pueda formar parte del código de un programa.
- **INTERFAZ**: Los títulos de botones, **OPCIONES** de menú y cualquier otro elemento relativo a la interfaz de Delphi u otra herramienta que esté describiéndose aparecerán en este estilo. Para indicar que se ha de abrir un menú o submenú y, a continuación, elegir una cierta opción se usará la notación **MENÚ—SUBMENÚ—OPCIÓN**.

Además de estos estilos, en el texto también se destacarán puntualmente párrafos completos con el objetivo de aclarar o advertir de algo importante:

> **NOTA**
>
> *Las notas se usan para destacar algo importante o aclarar algún concepto en relación con lo que está explicándose.*

Las referencias a ejercicios que puedan obtenerse desde el repositorio GitHub asociado al libro aparecerán en el texto como se muestra a continuación:

> **EJEMPLO 1.1** `NombreProyecto`
>
> Esta notación se usará para indicar la `carpeta` en la que se encuentra un cierto ejemplo y su nombre.

Finalmente, los fragmentos de código introducidos en el texto tendrán el aspecto siguiente:

```
1 procedure TForm1.FormShow(Sender: TObject);
2 var
3   unaVariable: String;
4 begin
5   with aNotification do begin
6     Name := 'Delphi y bases de datos'; // Aquí un comentario
7   end;
8 end;
```

Listado 1.1 Código de ejemplo

PARTE 1

ACCESO A DATOS LOCALES DESDE APLICACIONES NATIVAS

En esta primera parte del libro se introducirá el mecanismo fundamental de acceso a datos con que cuentan las últimas versiones de Delphi: **FireDAC**, usándolo en un contexto local desde aplicaciones nativas, ya sea ejecutándose en Windows, OS X, iOS o Android.

Comenzaremos desarrollando un ejercicio muy sencillo que nos permita tener una visión global, sin entrar en los pormenores, para después adentrarnos en

los fundamentos de FireDAC. A continuación, en los capítulos siguientes, nos familiarizaremos con las herramientas del entorno de Delphi que usaremos para desarrollar este tipo de aplicaciones: los módulos de datos, los componentes de interfaz vinculados a un origen de datos que servirán para diseñar la interfaz de usuario, gestores de bases de datos locales y embebidos, etc. Luego abordaremos el uso de IB ToGo e IBLite como gestor de datos embebido, especialmente para aplicaciones móviles. Terminaremos abordando algunos detalles relativos al uso de FireDAC con bases de datos de escritorio, bases de datos en memoria y el trabajo con datos Unicode.

Introducción a FireDAC **61**

Herramientas BDD en Delphi **89**

Interfaces de usuario con conexión a datos **123**

InterBase embebido **155**

Bases de datos de escritorio **181**

Bases de datos en memoria **209**

Bases de datos y Unicode **229**

Capítulo 2

INTRODUCCIÓN A FIREDAC

Al desarrollar una aplicación con Delphi podemos usar FireDAC en multitud de contextos diferentes. En los capítulos de este libro, según avancemos, tendremos ocasión de conocer varios de ellos. Este capítulo es nuestro punto de partida, en el que comenzaremos a fraguar los cimientos que nos permitirán ir construyendo proyectos cada vez más elaborados y complejos.

Nuestro primer objetivo es obtener una visión general del procedimiento que seguiremos habitualmente para trabajar con FireDAC. ¿Qué mejor modo de hacerlo que a través de un ejercicio práctico? Pongámonos manos a la obra colocando el primer pilar metafórico de los cimientos de los que hablábamos antes.

2.1 Hola FireDAC

Nos proponemos desarrollar el típico programa **Hola mundo** que, por tradición, se escribe siempre que se toma contacto por primera vez con un nuevo lenguaje o herramienta. En este caso, no obstante, el objetivo no es tan simple como la impresión de un mensaje en pantalla. Pretendemos crear una aplicación que muestre la información almacenada en una tabla de una base de datos, sin más funcionalidad por el momento. Para ello no precisaremos de ninguna herramienta externa ni de un servidor datos, todos los elementos que necesitamos están incluidos en Delphi.

> **NOTA**
>
> *A partir de este punto se asume que el lector tendrá instalada en su equipo de desarrollo una versión de Delphi que incluya FireDAC, a fin de poder llevar a cabo por sí mismo los pasos que se irán describiendo en los siguientes apartados de esta sección.*

2.1.1 Inicio del proyecto

Nuestro programa tiene como única finalidad familiarizarnos con la estructura de una aplicación simple en la que se utiliza FireDAC para acceder a una base de datos, por lo que bastará con que podamos ejecutarla en nuestro propio equipo de desarrollo. No obstante, si se tratase de un proyecto real probablemente nos interesase tener también como objetivo OS X, iOS y Android. Por ello vamos a iniciar un proyecto de tipo multi-dispositivo.

Usaremos la opción **NEW—MULTI-DEVICE APPLICATION** del menú **FILE**, tal y como se muestra en la Figura 2.1. Esto abrirá el cuadro de diálogo **MULTI-DEVICE APPLICATION**, ofreciéndonos las plantillas disponibles. Elegiremos la opción **BLANK APPLICATION** a fin de partir de un proyecto vacío. Este constará inicialmente de un formulario, la superficie en la que iríamos incluyendo los elementos de nuestra interfaz de usuario, que se abrirá de inmediato en el correspondiente diseñador.

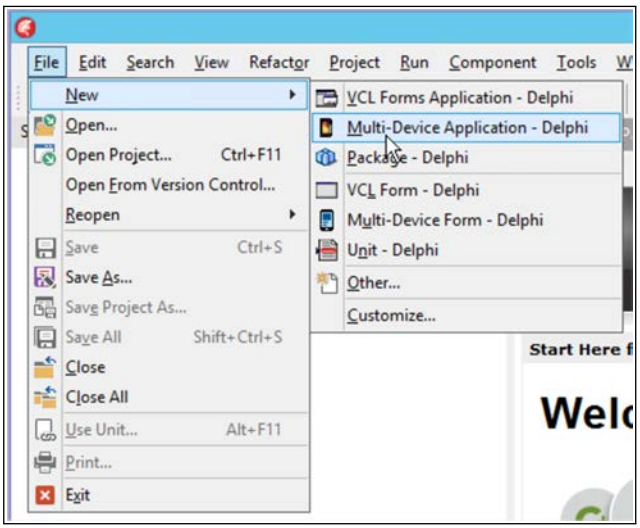


Figura 2.1 INICIAMOS UN NUEVO PROYECTO DE APLICACIÓN MULTI-DISPOSITIVO

En la ventana **PROJECT MANAGER**, el Gestor de proyectos de Delphi, podemos comprobar que ese formulario aparece como un módulo con extensión `.pas` y que tiene como hijo otro módulo, en este caso con extensión `.fmx`. El primero contendrá el código asociado al formulario y el segundo la definición de la interfaz propiamente dicha: lista de componentes y los valores asignados a sus propiedades.

> **NOTA**
>
> *Con la opción que hemos elegido para crear el proyecto estamos optando por utilizar la biblioteca de componentes **FMX**, también conocida como **Firemonkey**. El diseñador para este tipo de formularios, completamente multiplataforma, se denomina **FireUI**. La alternativa, con la que la aplicación únicamente podría ser compilada para Windows, sería el uso de la tradicional biblioteca VCL.*

2.1.2 Adición de un módulo de datos

Aunque podríamos añadir al formulario todos los componentes que vamos a necesitar, una buena práctica al desarrollar aplicaciones conectadas a bases de datos es aislar todos los elementos no visuales, aquellos cuya finalidad es facilitar el acceso a los datos, en un módulo independiente, concretamente un **módulo de datos**. Este es un contenedor, como los formularios, pero no cuenta con interfaz de usuario.

En general, para agregar cualquier nuevo elemento a un proyecto o nuevo componente a un contenedor usaremos **IDE INSIGHT**[1]. Para ello pulsamos **F6** y a continuación escribimos el inicio del nombre de lo que buscamos, en este caso `data module`, seleccionándolo de la lista de resultados de búsqueda (véase la Figura 2.2).

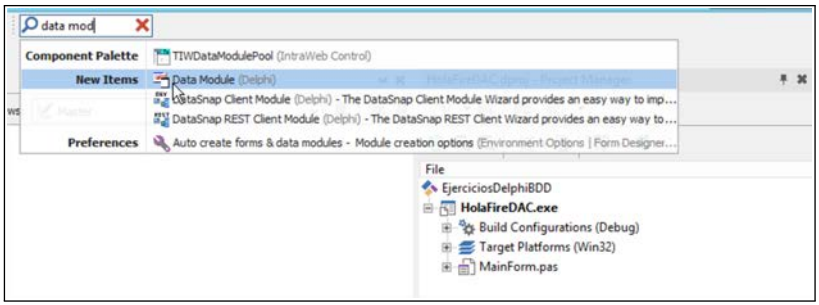


Figura 2.2 AÑADIMOS UN MÓDULO DE DATOS AL PROYECTO

El módulo de datos aparecerá abierto en el diseñador, completamente vacío. Lo usaremos en los siguientes apartados como recipiente para alojar los componentes FireDAC con los que crearemos y accederemos a una base de datos simple. Para ello comenzaremos agregando al módulo de datos un componente `TFDConnection`. Recuerda: pulsa **F6**, escribe el nombre del componente y pulsa **INTRO** para agregarlo al contenedor activo en el diseñador. También

[1]También podríamos ir al menú **FILE** para localizar esta misma opción, así como al menú contextual del proyecto en el Gestor de proyectos. Para añadir componentes a un contenedor podemos buscar tanto en la ventana **TOOL PALETTE** (la Paleta de componentes), aparece normalmente anclada en la parte inferior derecha del entorno, como en **IDE INSIGHT**.

puedes localizar el componente en la Paleta de componentes y hacer doble clic sobre él o arrastrarlo y soltarlo, lo que te resulte más cómodo.

EJEMPLO 2.1 `HolaFireDAC.dproj`

Puedes encontrar este proyecto completo en la carpeta `HolaFireDAC`. Para usarlo en tu equipo tendrás que editar la configuración del componente `TFDConnection` y corregir la ruta a la base de datos, usando aquella en la que hayas clonado el proyecto.

2.1.3 Configuración de la conexión

A continuación tendremos que configurar la conexión a la base de datos, estableciendo varias de las propiedades del componente `TFDConnection` recién añadido al módulo de datos. Podemos usar la ventana **OBJECT INSPECTOR** (el Inspector de objetos), localizando las propiedades que nos interesen y modificando su contenido, tal y como se muestra en la Figura 2.3.

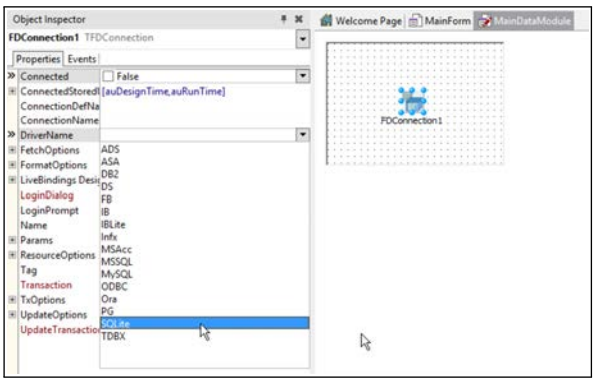


Figura 2.3 SELECCIONAMOS EL TIPO DE BASE DE DATOS

La alternativa consiste en abrir el menú contextual del componente, elegir la opción **CONNECTION EDITOR** y usar el editor mostrado en la Figura 2.4 para llevar a cabo la configuración.

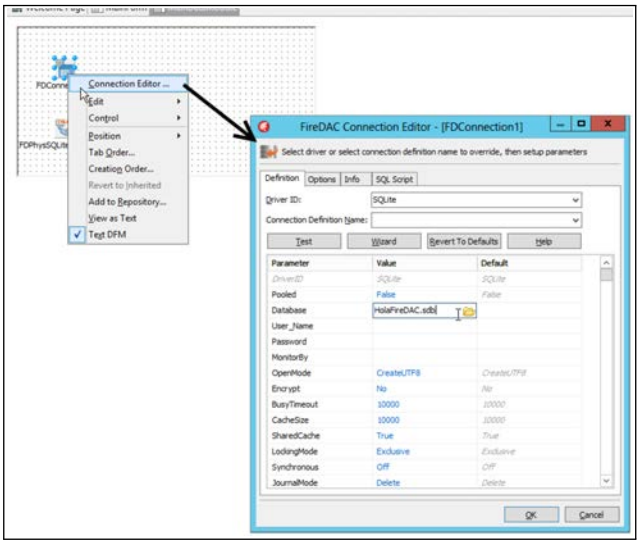


Figura 2.4 ABRIMOS EL EDITOR DE PARÁMETROS DE CONEXIÓN.

En ambos casos estableceremos las mismas propiedades, asignándoles los valores enumerados a continuación:

- `DriverName`: Esta propiedad establece el controlador FireDAC a utilizar. Tiene asociada una lista desplegable con todas las opciones disponibles en nuestro sistema. Elegiremos la opción `SQLite`.

- `Params.Database`[2]: Contiene el nombre de la base de datos y la ruta en que se encuentra. En este caso, al tratarse de una base de datos local alojada en un archivo, esa ruta será el camino a la carpeta en la que ya existe

[2]En el Inspector de objetos la propiedad `Params` tiene un simbolo **+** a su izquierda, con el que podemos desplegarla y acceder a sus subpropiedades. En la ventana **CONNECTION EDITOR** la propiedad aparece como `Database`.

o donde se creará la base de datos. Seleccionamos la carpeta donde tenemos el proyecto e introducimos `HolaFireDAC.sdb` como nombre de archivo.

- `LoginPrompt`: Mediante esta propiedad se indica a FireDAC si ha de solicitar credenciales de acceso a la base de datos o no. Por defecto está marcada, su valor es `True`. En este caso le asignaremos el valor `False`, desmarcándola, ya que no es preciso ese paso para acceder a una base de datos SQLite.

Una vez establecidas estas tres propiedades, procederemos a dar el valor `True` a la propiedad `Connected` para activar la conexión con la base de datos. Dato que esta no existe, la configuración por defecto del controlador para SQLite determina que se procederá a su creación (mira el valor por defecto de la propiedad `Params.OpenMode`). Lo que conseguimos, por tanto, es crear el archivo indicado por la propiedad `Params.Database`, inicialmente vacío pero ya preparado para operar sobre él como una base de datos SQLite.

> **NOTA**
>
> *Puedes abrir la ventana* **CONNECTION EDITOR** *y hacer clic en el botón* **TEST** *para comprobar que la conexión con la base de datos es satisfactoria.*

2.1.4 Creación de una tabla

Llegados a este punto tenemos una base de datos SQLite vacía. Para poder almacenar alguna información en ella antes es necesario definir, como mínimo, la estructura de una nueva tabla. Para ello reproduciremos los pasos indicados a continuación:

1. Abrimos la ventana **CONNECTION EDITOR** si no la teníamos abierta ya, usando para ello el menú contextual del componente `TFDConnection` como se apuntó antes.
2. Accedemos a la página **SQL SCRIPT** de dicha ventana y escribimos en la parte superior la sentencia SQL mostrada en el Listado 2.1, con la que crearemos una tabla que teóricamente nos servirá para guardar reservas de

asientos en algún tipo de espectáculo. Cada fila de la tabla contendrá un identificador único, un número de asiento y un nombre de cliente.

```
CREATE TABLE Entradas(
    id INTEGER PRIMARY KEY,
    asiento INTEGER,
    cliente CHAR(50)
);
```

Listado 2.1 Creación de una tabla

3. Hacemos clic en el botón **RUN DE SCRIPT** (véase la Figura 2.5) para ejecutar la sentencia. El resultado debería ser la obtención de un mensaje en la parte inferior comunicando que se ha ejecutado sin problemas.

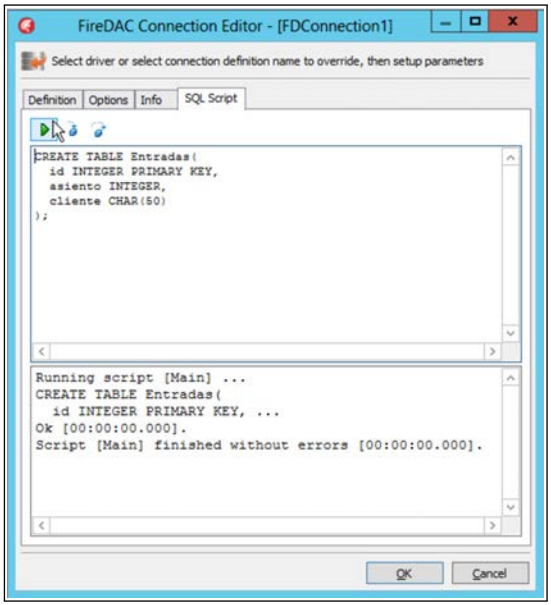


Figura 2.5 INTRODUCIMOS LA SENTENCIA SQL PARA CREAR LA TABLA

2.1.5 Introducción de datos

Para que la aplicación que estamos desarrollando pueda mostrar los datos de la tabla, esta habrá de contar con algún contenido. En este momento la tabla es una simple estructura, sin datos en su interior. Podríamos utilizar la página **SQL SCRIPT** de la ventana **CONNECTION EDITOR** para introducir las sentencias SQL necesarias para añadir los datos, tal y como hemos hecho para crear la tabla. No obstante, y con el objetivo de conocer otra herramienta fundamental del entorno de Delphi a la hora trabajar con bases de datos, vamos a emplear un método alternativo.

En la parte superior derecha del entorno, en el espacio que ocupa originalmente el Gestor de proyectos, encontraremos también la ventana **DATA EXPLORER** (el Explorador de datos). Partiendo de ella reproducimos los pasos siguientes:

> **NOTA**
>
> *Si no encuentras abierta en el entorno algunas de las ventanas mencionadas en el texto, ve al menú principal de Delphi y utiliza las opciones del menú* **VIEW** *para hacerlas visibles.*

1. Localiza en el Explorador de datos el nodo **SQLITE DATABASE**, abre su menú contextual y elige la única opción disponible: **ADD NEW CONNECTION**.
2. Se abre la ventana **CONNECTION EDITOR**, en la que únicamente tendremos que hacer clic en la propiedad `Database` y facilitar el camino a la base de datos. Podemos usar el botón **TEST** para probar la conexión antes de usar el botón **OK** para cerrar la ventana.
3. Abrimos el nodo agregado al Explorador de datos, localizamos la tabla que habíamos creado anteriormente, abrimos su menú contextual y elegimos la opción **VIEW**, tal y como se muestra en la Figura 2.6.
4. Se abre en la parte central del entorno un editor con la estructura de la tabla y sin contenido. Podemos introducir unas filas de datos de ejemplo, como se ha hecho en la Figura 2.7, antes de cerrar esta página y terminar el proceso.

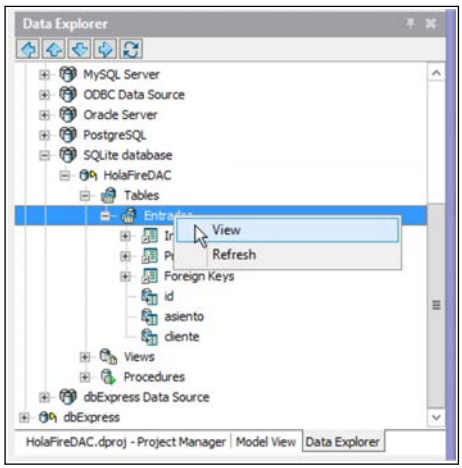


Figura 2.6 ABRIMOS LA TABLA DESDE EL EXPLORADOR DE DATOS

Figura 2.7 INTRODUCIMOS ALGUNOS DATOS EN LA TABLA

En este momento nuestra base de datos, con una única tabla, ya tiene algo de contenido. Es todo lo que necesitamos para poder avanzar en el desarrollo de nuestro proyecto.

2.1.6 Otros componentes a añadir al módulo de datos

Además del `TFDConnection`, mediante el que hemos creado la base de datos y que facilitará a la aplicación la conexión con la misma, hemos de añadir al módulo de datos otros tres componentes. Como antes, podemos pulsar **F6** e introducir su nombre para localizarlos y agregarlos. Estos componentes son:

- `TFDTable`: Usaremos este componente para obtener todo el contenido de la única tabla existente en la base de datos. Al agregarlo al formulario de datos se establecerá automáticamente un vínculo con el `TDFConnection`, pero podemos modificarlo cambiando el valor de la propiedad `Connection` del `TDFTable`. A continuación abriremos la lista asociada a la propiedad `TableName`, eligiendo la opción existente: `Entradas`. Finalmente daremos el valor `True` a la propiedad `Active`, habilitando la conexión del componente con la base de datos y la recuperación del contenido de la tabla.

- `TFDPhysSQLiteDriverLink` y `TFDGUIxWaitCursor`: Al añadir estos componentes al módulo incluiremos en el ejecutable el código necesario para trabajar con bases de datos SQLite, sin necesidad de ningún controlador externo, así como los elementos de interfaz de usuario que usa FireDAC para notificar, por ejemplo, que hay un proceso en curso. No es necesario editar ninguna propiedad de estos componentes.

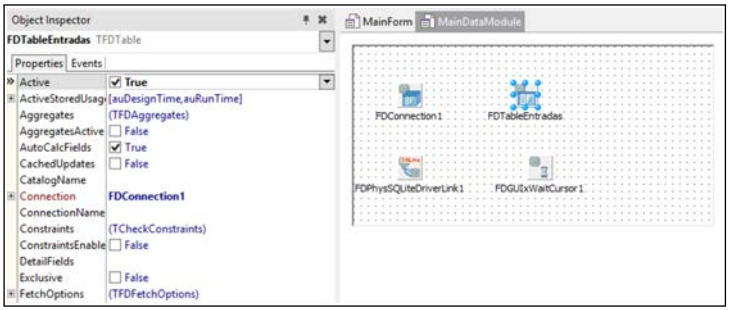


Figura 2.8 COMPONENTES INTRODUCIDOS EN EL MÓDULO DE DATOS

2.1.7 Diseño de la interfaz de usuario

A fin de visualizar la información que antes añadíamos a la tabla de nuestra base de datos deberemos agregar al formulario, inicialmente vacío, los controles apropiados. En este caso vamos a optar por la sencillez, ya que únicamente se pretende mostrar algunos datos, sin opciones de navegación, modificación o búsqueda.

Añade al formulario un control `TListView`. Para que ocupe todo el espacio disponible modifica su propiedad `Align`, dándole el valor `Client`. Cambia también la propiedad `ItemAppearance`, como se muestra en la Figura 2.9, seleccionando la opción `ListItemRightDetail`. De esta forma en la lista se mostrará un texto principal e información de detalle a la derecha.

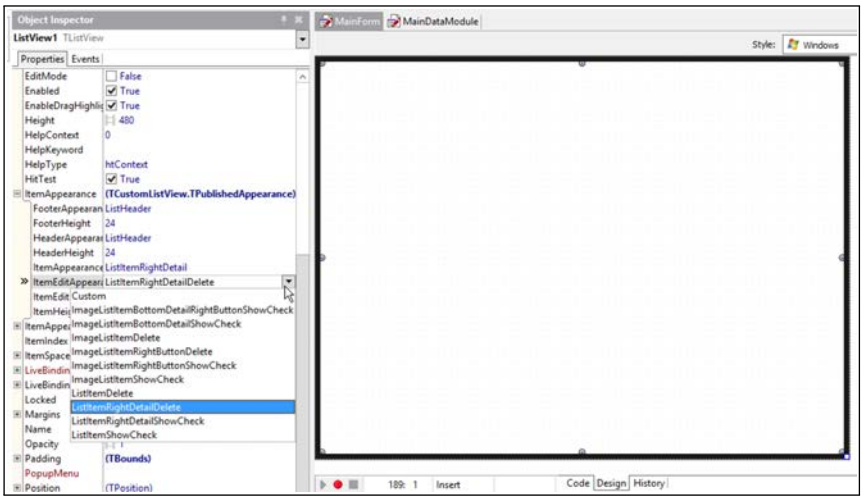


Figura 2.9 CONFIGURAMOS EL TLISTVIEW

Esta sería toda nuestra interfaz de usuario. Ahora, para que el `TListView` muestre el contenido de la tabla, hemos de conectar este componente con el `TDFTable` que teníamos en el módulo de datos. Para ello se sigue el proceso descrito en los siguientes apartados.

2.1.8 Enlace entre interfaz y datos

La parte final en el desarrollo de este proyecto será la creación de un vínculo entre la información alojada en la base de datos, que ya tenemos seleccionada en el componente `TFDTable` del módulo de datos, y la interfaz de usuario, concretamente el componente `TListView`.

Lo primero que hemos hacer es agregar al formulario una referencia al módulo de datos, de tal forma que el primero pueda usar los elementos incluidos en el segundo. Esta acción la podemos llevar a cabo de múltiples formas. Una de las más inmediatas consiste en elegir la opción **FILE—USE UNIT** (véase la Figura 2.10). Esta da paso a un cuadro de diálogo con la lista de módulos disponibles, en nuestro caso solamente hay uno, del que podemos elegir el que nos interese. Este paso[3] solo hay que darlo una vez, después podríamos enlazar tantos controles a componentes de datos como necesitásemos.

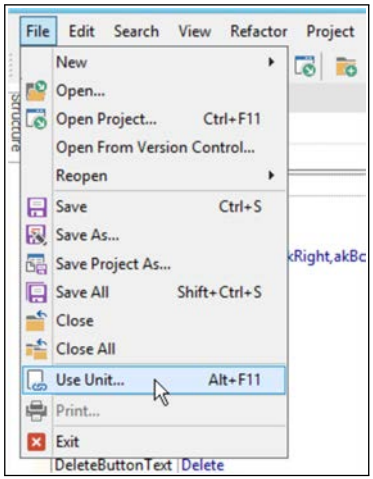


Figura 2.10 AGREGAMOS AL FORMULARIO UNA REFERENCIA AL MÓDULO DE DATOS

[3]Lo que hace la opción **USE UNIT** es agregar a la cláusula `uses` del módulo actual el nombre de otro módulo del proyecto, de forma que desde el primero es posible hacer referencia a elementos del segundo.

A continuación vamos a abrir el **LIVE BINDINGS DESIGNER**, la herramienta que nos permitirá crear visualmente los vínculos entre datos y controles. Para ello abrimos el menú contextual del `TListView` y usamos la opción **BIND VISUALLY**. La citada herramienta se abrirá como un panel en la parte inferior del entorno, debajo del diseñador de formularios. En ella podremos ver una pequeña tabla con el nombre del `TListView` como título, y una lista de los campos que es posible vincular, y otra que representa al `TFDTable` que tenemos en el módulo de datos. En la Figura 2.11 la primera aparece a la izquierda y la segunda a la derecha.

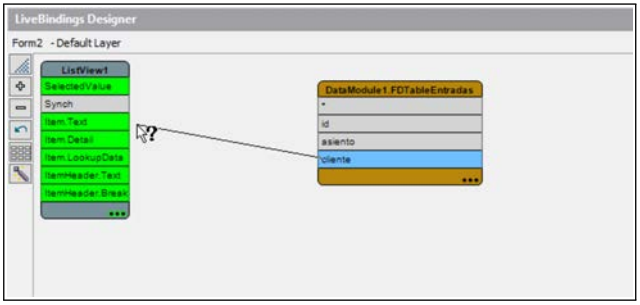


Figura 2.11 ENLAZAMOS COLUMNAS DE LA TABLA CON PROPIEDADES DEL CONTROL

Mediante la técnica de arrastrar y soltar, como se aprecia en la misma Figura 2.11, conectaremos la columna `cliente` de la tabla `entradas` con la propiedad `Item.Text` del `TListView`, y la columna `asiento` con la propiedad `Item.Detail`. Completado este paso, las conexiones deberían ser las representadas en la Figura 2.12.

Al establecer el primero de los vínculos comprobaremos cómo se añaden automáticamente al formulario dos nuevos componentes: un `TBindSourceDB` y un `TBindingsLists`. Además en el diseñador de enlaces la tabla que representaba al `TFDTable` habrá sido sustituida por otra que corresponde al `TBindSourceDB`. Este actúa como intermediario entre el `TListView` y el `TDFTable`.

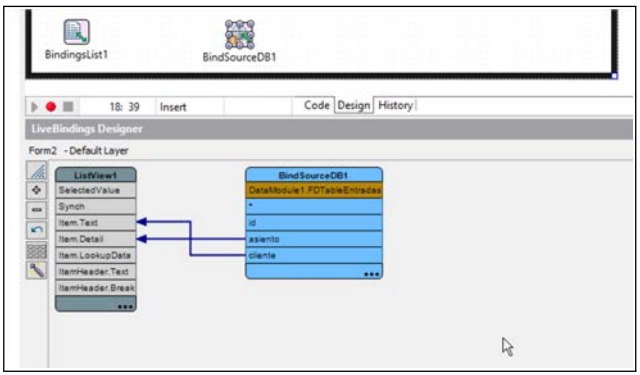


Figura 2.12 CONFIGURACIÓN FINAL DE LOS ENLACES

2.1.9 Prueba de la aplicación

Si hemos completado correctamente todos los pasos indicados en los apartados previos, en el propio diseñador de formularios deberíamos estar viendo, dentro del control `TListView`, los datos que habíamos introducido anteriormente en la tabla.

Bastará con que pulsemos **F9** para compilar y ejecutar el programa, cuyo resultado debería ser similar al mostrado en la Figura 2.13. Por cada fila podremos ver el nombre del cliente y el asiento asignado. Si el formulario apareciese vacío, o se generase algún error al ejecutar el proyecto, asegúrate de que la configuración del componente `TFDConnection` es correcta, especialmente la ruta a la base de datos, y de que la propiedad `Active` del componente `TFDTable` tiene el valor `True`.

Lo que hemos conseguido es que una aplicación recupere información almacenada en una base de datos, en este caso una tabla completa, y la muestre en una sencilla interfaz de usuario. Para ello no hemos escrito una sola línea de código (con la excepción del SQL para crear la tabla), ha bastado con configurar adecuadamente apenas media docena de componentes. En las siguientes secciones de este capítulo ahondaremos un poco más en varios de esos componentes y, en general, la forma de utilizar FireDAC para operar con bases de datos locales.

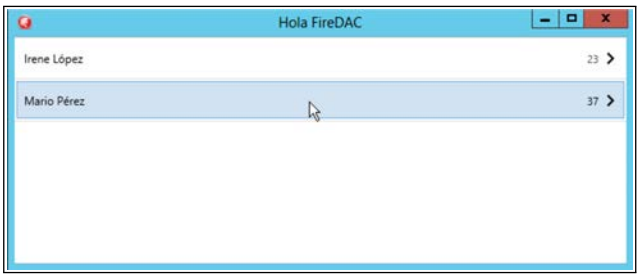


Figura 2.13 LA APLICACIÓN EN FUNCIONAMIENTO

> **NOTA**
>
> *Si quieres mostrar también el identificador que corresponde a cada supuesto cliente, usa el* **LIVE BINDINGS DESIGNER** *para conectar la columna* `id` *de la tabla con la propiedad* `ItemHeader.Text` *del* `TListView`.

2.2 Bases de datos locales y FireDAC

Ahora que hemos satisfecho nuestra lógica curiosidad inicial sobre cómo utilizar FireDAC en una aplicación propia, con el desarrollo del sencillo ejercicio de la sección previa, vamos a ocuparnos de aspectos algo más teóricos, comenzando por las posibilidades que tendremos a nuestro alcance al desarrollar proyectos que precisan almacenamiento local de datos.

Entendemos por base de datos **local** aquella que se aloja en el mismo dispositivo que ejecutará el programa, sea este un ordenador de escritorio o portátil, una tableta, un teléfono móvil, un *smart watch* o cualquier otro aparato móvil que ejecute Windows, OS X, iOS o Android. Esto implica que no es precisa conexión hacia el exterior a la hora de obtener y manipular los datos, únicamente hay que abrir un archivo, almacenado en el propio dispositivo, y operar sobre él.

Dependiendo de la opción que elijamos, de las que van a describirse en los siguientes apartados de esta sección, será la propia aplicación la que gestione el archivo que contiene la información o bien delegará dicha gestión en otro programa. La elección que hagamos también influirá en otros aspectos, como los elementos que será necesario distribuir al desplegar el proyecto para ponerlo en explotación.

Los siguientes apartados describen las opciones que tenemos a nuestro alcance usando la versión de FireDAC que se incluye en la edición *Professional* de Delphi.

2.2.1 InterBase Lite e InterBase ToGo

InterBase es un RDBMS completo, contemplando aspectos como la definición de procedimientos almacenados y funciones, el diseño de vistas, la comprobación de integridad referencial, etc. A pesar de ello no precisa muchos recursos para su funcionamiento, de ahí que existan versiones de este producto capaces de funcionar en dispositivos móviles.

InterBase Lite e InterBase ToGo son ediciones de este RDBMS pensadas para su uso embebido, como gestores de bases de datos locales. En ambos casos no se precisa instalación, solamente hay que redistribuir los archivos apropiados con nuestra aplicación, y son gestores de datos para un único usuario. La principal diferencia entre estas dos opciones es que la versión Lite es gratuita, pero únicamente puede utilizarse sobre bases de datos de tamaño reducido, de 100 megabytes como máximo, y su motor se ejecuta únicamente en un núcleo de procesamiento. Además solamente se permite una conexión simultánea con la base de datos.

Las ventajas de InterBase ToGo sobre la versión Lite son múltiples. El tamaño de la base de datos no está limitada de antemano. A pesar de que también es un RDBMS monousuario, la aplicación puede tener habilitadas hasta 8 conexiones simultáneas con bases de datos. Además se aprovechan las arquitecturas multinúcleo con que cuentan actualmente tanto ordenadores como tabletas y teléfonos móviles, usando hasta 4 núcleos de procesamiento simultáneamente. Aparte de todas las anteriores, seguramente la diferencia más importante sea que la edición ToGo nos permite cifrar la base de datos completa tanto con AES (*Advanced Encription Standard*) como con DES (*Data Encryption Standard*), asegurando así la confidencialidad de la información almacenada en ella.

> **NOTA**
>
> *InterBase Lite e InterBase ToGo pueden ejecutarse sobre Windows, OS X, iOS y Android. Podemos encontrar más información sobre estas versiones de InterBase en* `http://www.embarcadero.com/es/products/interbase`*. Las instrucciones sobre cómo obtener la licencia de uso de InterBase Lite se encuentra en* `http://www.embarcadero.com/ibinstructions`*. En el capítulo 5 aprenderemos más sobre el uso de InterBase embebido.*

Optar desde un principio por InterBase como base de datos local o embebida facilitaría una posterior transición, en caso de que el volumen de información a almacenar así lo requiriese, a la versión de escritorio o servidor. El formato de archivo usado por las ediciones Lite y ToGo es el mismo, por lo que el traslado de la información al nuevo entorno resultaría sencillo.

2.2.2 SQLite

Otra opción de tipo embebido y multiplataforma para el almacenamiento de datos es SQLite, el motor de bases de datos que usábamos en el ejercicio desarrollado en la sección previa. A diferencia de InterBase, SQLite es un producto de software libre, con el código fuente a disposición pública y que no requiere ningún tipo de licencia para su uso y redistribución. Esto ha hecho que sea el motor para operar sobre bases de datos locales con SQL más difundido, contando con interfaces de acceso desde múltiples lenguajes de programación.

SQLite contempla el uso de múltiples tablas, vistas e índices, todo ello almacenado en un único archivo alojado en el mismo dispositivo que ejecuta la aplicación. El formato de dicho archivo es **multiplataforma**, no hay diferencias entre una base de datos SQLite en iOS respecto a Android o en OS X respecto a Windows. Es factible copiar el archivo que contiene la base de datos de un sistema operativo a otro y utilizarlo sin más. No obstante la mayor ventaja de SQLite la encontramos en los mínimos recursos que precisa para su funcionamiento, algo que junto al reducido tamaño de la biblioteca hace posible usar SQLite demandando apenas un megabyte de memoria. Por tanto SQLite es

una solución ideal para dispositivos pequeños y con pocos recursos, como los *wearables*.

Además de la portabilidad y reducida exigencia de recursos, SQLite también ofrece un motor SQL completo, ajustado al estándar SQL92, y contempla el uso de **transacciones**, con tolerancia a fallos como caídas del sistema. No hay un límite preestablecido en el tamaño de las bases de datos, depende del espacio disponible y los límites del sistema de archivos, pudiendo alcanzar el orden de los terabytes. Si el rendimiento es un factor determinante en el proyecto que estemos desarrollando, SQLite puede operar completamente en memoria mejorando la **velocidad de acceso** a los datos respecto a soluciones basadas en archivos y, por supuesto, aquellas que precisan la comunicación con un servidor.

Entre las limitaciones de SQLite hay que destacar el hecho de que es una alternativa monousuario, no contemplándose la existencia de usuarios y roles distintos en una base de datos ni el acceso concurrente a la información. Si bien es posible extender SQLite mediante su API propia, no existen las funciones y procedimientos almacenados habituales en los RDBMS.

> **NOTA**
>
> *Como pudimos comprobar en el ejercicio desarrollado en la sección previa, es posible trabajar con bases de datos SQLite desde una aplicación Delphi sin necesidad de instalar software adicional alguno.*

2.2.3 Microsoft Access

Access es probablemente la aplicación de gestión de bases de datos de escritorio más difundida en entornos de tipo SOHO (*Small-Office, Home-Office*). Se trata de un producto específico para Windows, por lo que su uso está vinculado a ordenadores personales. La cantidad de información almacenada a día de hoy en bases de datos Access es incalculable, razón por la que es probable que en algún momento precisemos que una aplicación que estemos desarrollando use dicho formato. Asimismo, el almacenamiento de información en una base de datos Access abre la puerta a que el usuario pueda acceder a ella por una vía alternativa, con la aplicación Microsoft Access en lugar del programa específico que se haya desarrollado para dicha tarea.

Al igual que en los casos anteriores, una base de datos Access se almacena en un único archivo alojado en el mismo sistema que ejecuta la aplicación. Access no es un RDBMS, por lo que hay muchas características habituales de SQL que no están disponibles. Asimismo cuenta con objetos adicionales que no encontraremos en un RDBMS, como son los formularios o los informes, almacenados también en la base de datos.

El principal inconveniente de las bases de datos Access es que son una solución limitada a Windows, no siendo posible su uso en OS X, iOS o Android desde Delphi. Si trabajamos en un proyecto multiplataforma, por tanto, es una alternativa que descartaríamos de partida.

> **NOTA**
>
> *La actual versión de FireDAC facilita el acceso a bases de datos Access a través de los controles ODBC de Microsoft, pudiendo utilizarse tanto bases de datos* `.mdb` *como* `.accdb` *correspondientes a las versiones 95 a 2010. La más reciente versión 2013 no está soportada.*

2.2.4 Archivos de datos en otros formatos

En ocasiones las necesidades de tratamiento de datos de una aplicación pueden ser tan simples que incluso una base de datos embebida o de escritorio, como las citadas en los apartados previos, puede resultar excesivo, siendo suficiente el almacenamiento directo desde la aplicación a un archivo simple. El formato de dicho archivo puede ser propietario, pero es recomendable emplear algún formato estándar como puede ser CSV (*Comma Separated Values*), XML (*Extensible Markup Language*) o DIF (*Data Interchange Format*).

Cargar en memoria datos guardados en un archivo con cualquiera de esos formatos no representa un gran obstáculo, pero una vez completada esta operación surge la necesidad de operar sobre dichos datos. ¿Cómo efectuar consultas sobre información que no está gestionada por un motor SQL? FireDAC facilita esta tarea gracias a componentes como `TFDMemTable` y `TFDLocalSQL`. El primero actúa como una **base de datos en memoria**, cuya estructura es posible definir libremente o bien generarla a partir de archivos de datos existentes. El

segundo facilita la ejecución de consultas SQL sobre conjuntos de datos que no están vinculados realmente a una base de datos, como sería este caso.

Un proyecto que use esta alternativa será completamente multiplataforma, ya que únicamente usará archivos de texto que emplean un cierto separador (CSV/DIF) o unas etiquetas (XML), sin depender de ningún motor de gestión de datos ni biblioteca externa. El código de los componentes FireDAC se integra en el propio ejecutable, encargándose de todas las operaciones que necesite la aplicación. A cambio se renuncia a ciertos beneficios, como la posibilidad de usar transacciones (soportadas por InterBase Lite/ToGo y SQLite) que garanticen la integridad de los datos, o el hecho de tener a disposición una ruta fácil de transición a un RDBMS más potente, como ocurre con InterBase.

NOTA

En el capítulo 7 aprenderemos a construir una aplicación siguiendo este enfoque, utilizando solamente componentes FireDAC para gestionar los datos.

2.3 Estructura de una aplicación que usa FireDAC

Las aplicaciones Delphi que utilizan componentes FireDAC, siempre circunscribiéndonos a un entorno local, suelen tener una estructura similar, con independencia del tipo de base de datos concreta que se utilice. Esta estructura es, además, la base para aplicaciones cliente/servidor o distribuidas, en las que simplemente se agregan algunos elementos adicionales. La Figura 2.14 es un esquema de bloques representando dicha estructura. Distinguimos tres capas:

- **Interfaz de usuario**: Podemos utilizar tanto la biblioteca VCL como la FMX para diseñar la interfaz de usuario de la aplicación, según nuestras preferencias y las plataformas objetivo a las que se dirija el proyecto. A lo largo de este libro utilizaremos preferentemente la FMX, pero la esencia de los procedimientos a seguir sería fundamentalmente la misma con la VCL.

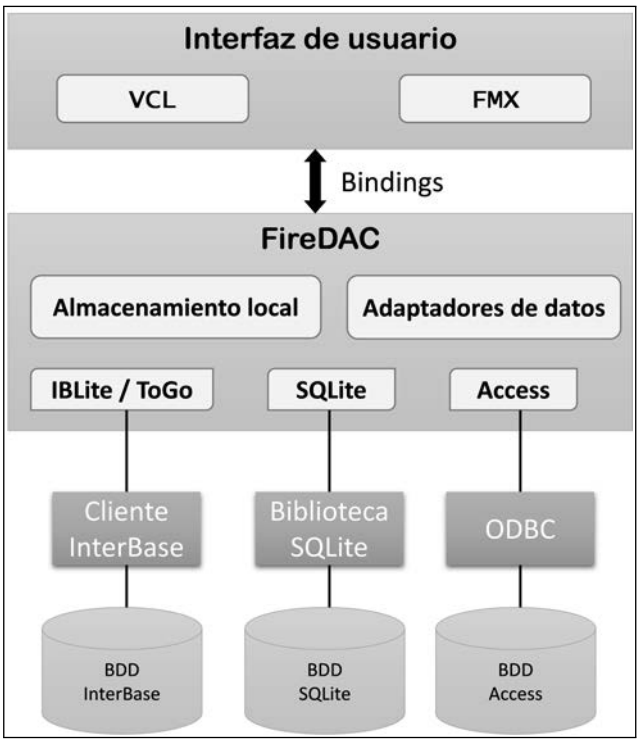


Figura 2.14 ESTRUCTURA DE UNA APLICACIÓN DELPHI QUE USA FIREDAC PARA ACCEDER A UNA BASE DE DATOS LOCAL

- **FireDAC**: En esta capa están todos los componentes relativos a gestión de datos, incluyendo los que conectan con la base de datos (específicos según el producto con el que vaya a trabajarse), los que actúan como intermediarios entre la conexión y el almacenamiento local (adaptadores de datos), etc.

- **Software cliente**: En la parte inferior del esquema tenemos las bibliotecas cliente que se ocupan del acceso físico al archivo que aloja la base de datos, un software cliente que sí es necesario redistribuir con la aplicación como lo ha sido siempre (usando otras tecnologías de acceso a datos de Delphi).

Todos los componentes FireDAC pasarán a formar parte de la aplicación una vez se haya compilado, incluyendo los controladores FireDAC para cada base de datos. Esto representa una diferencia clara respecto a BDE o dbExpress, bibliotecas que nos obligaban a redistribuir los controladores apropiados a cada caso. Con FireDAC el código del controlador se integra en nuestro ejecutable.

> **NOTA**
>
> *El software cliente de SQLite puede en algunos casos enlazarse estáticamente en el ejecutable de la aplicación, eliminando la necesidad de redistribuirlo por separado.*

2.3.1 Controladores FireDAC

Los controladores FireDAC para acceso a distintos tipos de orígenes de datos los encontraremos en la página **FIREDAC LINKS** de la Paleta de herramientas (véase la Figura 2.15). Concretamente son todos los componentes cuyo nombre se ajusta al patrón `TFDPhys`**`TIPO`**`DriverLink`, por ejemplo el componente `TFDPhysSQLiteDriverLink` que usábamos en el proyecto desarrollado en la primera sección.

> **NOTA**
>
> *La lista de controladores disponibles en la mencionada página de la Paleta de componentes dependerá de la edición de Delphi con que estemos trabajando. En la edición Professional no encontraríamos, por ejemplo, controladores como `TFDPhysDB2DriverLink` o `TFDPhysOracleDriverLink`.*

Cuando insertamos uno de esos componentes en un formulario o módulo de datos de nuestro proyecto, lo que conseguimos es añadir a la cláusula `uses` del módulo correspondiente una referencia a un módulo cuyo nombre se ajusta al patrón `FireDAC.Phys.TIPO`, por ejemplo `FireDAC.Phys.SQLite`. Dicho módulo contiene el código fuente del controlador FireDAC para una base

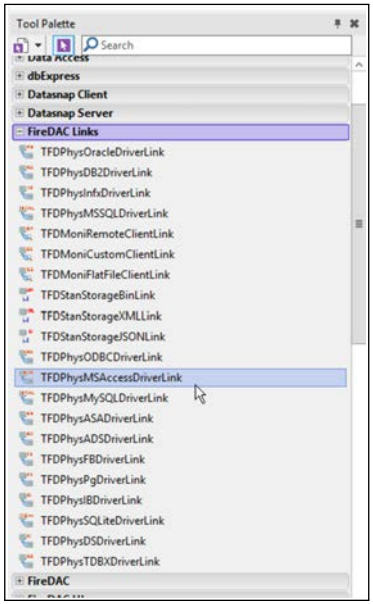


Figura 2.15 COMPONENTES PARA ENLAZAR LOS CONTROLADORES EN EL PROYECTO

de datos concreta. De hecho, una vez se ha añadido la mencionada referencia podríamos eliminar el componente y el proyecto compilaría y funcionaría igualmente. Estos componentes son sencillamente un atajo que nos ahorran agregar la referencia manualmente.

Entre los controladores FireDAC podemos encontrar dos que actúan a modo de puente o pasarela, facilitando el acceso a cualquier origen de datos ODBC (`TFDPhysODBCDriverLink`) o bien usar cualquier controlador dbExpress (`TFDPhysTDBXDriverLink`). De esta forma se amplia la cantidad de bases de datos a las que es posible acceder desde una aplicación Delphi mediante FireDAC.

Los servicios ofrecidos por el controlador que hayamos elegido permitirán al componente `TFDConnection` establecer conexión con el software cliente de la base de datos, ya sea directamente o bien a través de una infraestructura de red cuando se trabaja con un servidor. Habilitada la conexión la aplicación

podrá recuperar conjuntos de datos, así como ejecutar otro tipo de sentencias SQL, siempre según las características de la base de datos con la que se esté trabajando.

2.3.2 Conjuntos de datos

Disponiendo de una conexión a la base de datos, en una capa que estaría justo por encima de la formada por los controladores FireDAC y el componente de conexión, encontramos los componentes que representan conjuntos de datos. La clase base para conjuntos de datos FireDAC es `TFDDataSet`, derivada de la clase genérica `TDataSet`[4]. De `TFDDataSet` derivan, entre otras, tres clases que corresponden a los tres componentes fundamentales que usaremos al trabajar con FireDAC:

- **`TFDTable`**: Representa un conjunto de datos obtenido a partir del contenido de una tabla de la base de datos. Cuenta con la funcionalidad necesaria para introducir cambios en los datos y trasladarlos a la base de datos.

- **`TFDQuery`**: Mediante este componente es posible ejecutar cualquier consulta SQL, obteniendo un conjunto de resultados que puede proceder de múltiples tablas. Al igual que el anterior cuenta con las funciones necesarias para manipular los datos y transferir los cambios a la base de datos.

- **`TFDMemTable`**: Este componente hace posible trabajar con un conjunto de datos alojado en memoria, siendo opcional el uso de la conexión a la base de datos para obtener información o comunicar cambios.

> **NOTA**
>
> *Encontraremos los tres componentes previos en la página* **FIREDAC** *de la Paleta de componentes.*

[4] `TDataSet` es una clase que actúa como conjunto de datos genérico, siendo la base de los conjuntos de datos de IBX (`TIBCustomDataSet`), de dbExpress (`TCustomSQLDataSet`), etc.

Las tres clases anteriores están derivadas de `TFDAdaptedDataSet`. Esta clase implementa las funciones que hacen posible la ejecución de consultas, obtención de datos y envío de cambios a orígenes heterogéneos, usando para ello el adaptador apropiado. Esta es una funcionalidad que, por tanto, heredan los citados componentes.

Además de para trabajar con conjuntos de datos, en la página **FIREDAC** también encontraremos componentes cuya finalidad es facilitar la ejecución de consultas SQL en general (`TFDCommand`), la gestión de transacciones cuando se opera con un RDBMS (`TFDTransaction`), la ejecución de procedimientos almacenados (`TFDStoredProc`), etc.

2.3.3 Elementos de interfaz

Además de componentes no visibles cuya finalidad es encargarse de la gestión de conexiones y datos, como todos los citados hasta ahora, FireDAC también cuenta con algunos controles con una parte visual (véase la Figura 2.16). Estos elementos pasan a formar parte de la interfaz de usuario de la aplicación, por ejemplo a fin de poder solicitar las credenciales de conexión a una base de datos o indicar visualmente que hay una tarea en espera de ser completada.

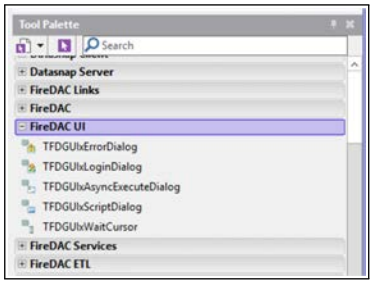


Figura 2.16 COMPONENTES FIREDAC ASOCIADOS A ELEMENTOS DE LA INTERFAZ DE USUARIO.

El único componente de este tipo que obligatoriamente hemos de agregar a nuestros proyectos es `TFDGUIxWaitCursor`. Este es usado por FireDAC para mostrar un cursor de espera.

Todos los componentes de este grupo cuentan con tres implementaciones distintas, recurriéndose a una u otra según que nuestra aplicación use la FMX, la VCL o sea una aplicación de consola. La propiedad `Provider` permite cambiar el tipo, tal y como se muestra en la Figura 2.17. Este cambio afectaría a todos los componentes de interfaz de FireDAC.

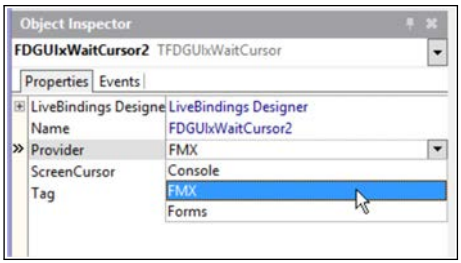


Figura 2.17 SELECCIÓN DEL PROVEEDOR DE INTERFAZ DE USUARIO

Al igual que ocurría con los controladores FireDAC, estos componentes lo que hacen es agregar una referencia al módulo adecuado en la cláusula `uses` de nuestro formulario o módulo de datos. Tras esta operación podríamos eliminarlos sin ningún problema.

2.3.4 Otros componentes FireDAC

A pesar de que FireDAC es una solución universal de acceso a datos, permitiéndonos usar los mismos componentes sobre distintos tipos de bases de datos, también incorpora componentes más específicos, diseñados para aprovechar las ventajas de productos concretos. Encontramos dichos componentes en la página **FIREDAC SERVICES** de la Paleta de componentes.

La nomenclatura de estos componentes nos indica tanto el tipo de base de datos sobre la que operan como el tipo de servicio que ofrecen. Así, mediante el componente `TFDSQLiteBackup` es posible realizar una copia de seguridad de una base de datos SQLite y restaurarla, con el componente `TFDIBConfig` se facilita el acceso a los parámetros de configuración de bases de datos InterBase, y con el componente `TFDMSAccessService` las funciones de compactación y reparación de bases de datos Access podrían llevarse a cabo desde una aplicación Delphi.

2.4 Resumen

Al finalizar este capítulo, en el que se ha desarrollado un proyecto básico con FireDAC y se ha hecho un recorrido posterior por la estructura de una aplicación que use dichos componentes, tenemos una visión general de lo que nos ofrece esta tecnología de acceso a datos. Nos hemos concentrado sobre todo en los servicios relacionados con el trabajo con orígenes de datos locales, pero la arquitectura fundamental sería la misma en un contexto cliente/servidor o distribuido.

Con este conocimiento de FireDAC, por ahora relativamente superficial, estamos en disposición de comenzar a profundizar en aspectos cada vez más concretos. Ese será el objetivo de capítulos posteriores. Antes, sin embargo, es necesario que conozcamos las herramientas indispensables para trabajo con bases de datos con que cuenta el entorno de Delphi. Este será el objetivo del capítulo siguiente.

Capítulo 3

HERRAMIENTAS DEL ENTORNO DE DELPHI PARA TRABAJAR CON BASES DE DATOS

Como bien sabemos Delphi es un entorno de desarrollo de tipo RAD (*Rapid Application Development*), lo cual significa que muchas de las tareas que es necesario llevar a cabo durante la creación de un proyecto pueden ser completadas visualmente, ya sea mediante herramientas integradas en el entorno o arrastrando y soltando componentes que después son personalizados mediante la modificación de sus propiedades.

El IDE de Delphi cuenta con varias utilidades que facilitarán nuestro trabajo a la hora de conectar nuestra aplicación con bases de datos. El objetivo de este capıtulo es que nos familiaricemos con esas herramientas, a fin de que seamos

capaces de aprovecharlas lo mejor posible en los capítulos siguientes, cuando abordemos el diseño de proyectos concretos. Algunas de estas herramientas, como los módulos de datos o el Explorador de datos, ya los hemos usado en el capítulo previo.

3.1 Módulos de datos

Cuando se comienza a trabajar en el desarrollo de una aplicación Delphi que va a conectar a bases de datos es bastante habitual, incluso por parte de usuarios ya experimentados en la creación de proyectos con Delphi, el ir agregando al formulario en el que se trabaja los componentes de conexión a datos a medida que van necesitándose. Esta costumbre no es recomendable ya que, por regla general, tarde o temprano nos llevará a agregar los mismos componentes con la misma configuración a otros formularios en los que también se precise la misma información.

Más allá del hecho de duplicar componentes y configuración, lo cual de por sí ya puede complicar la posterior fase de mantenimiento en caso de que hubiese que aplicar cambios en los parámetros de conexión a la base de datos, sin duda el mayor inconveniente es que también se duplicará código relacionado exclusivamente con la gestión de datos. A largo plazo esto será una fuente de problemas, especialmente si la persona que ha de mantener el proyecto no es la misma que lo desarrolló.

Para evitarlo existen los **módulos de datos**, un tipo de contenedor que utilizábamos en el ejercicio propuesto en el capítulo previo. Un mismo proyecto puede contar con tantos módulos de datos como se precisen. Habitualmente sólo usaríamos uno, pero si el proyecto lo requiere, porque conecte con múltiples bases de datos, podría ser recomendable emplear varios.

Los módulos de datos son clases derivadas del tipo `TDataModule` que, a su vez, es un descendiente directo de la clase `TComponent`, raíz de todos los componentes Delphi. Un módulo de datos, por tanto, es idéntico en una aplicación basada en la VCL que en otra que use la FMX, razón por la que únicamente puede alojar componentes no visuales, como son los que forman parte de FireDAC.

Centralizar los componentes FireDAC y todo el código asociado en un módulo de datos es una inversión que realizamos durante el desarrollo inicial cuyos frutos obtendremos en el futuro, siempre que tengamos que realizar cualquier tipo de actualización o cambio, ya que estaremos completamente seguros de que solamente hay un punto a revisar: el contenido del módulo de datos.

3.2 Data Explorer

El Explorador de datos (ventana **DATA EXPLORER**) lo usábamos en el capítulo previo para acceder a la base de datos SQLite de ejemplo, examinando y modificando su contenido. En su interior encontraremos dos nodos principales llamados **FIREDAC** y **DBEXPRESS**, cada uno de los cuales contendrá tantos subnodos como controladores haya disponibles. El nodo de cada controlador puede estar vacío, si no hay definida ninguna conexión que lo use, o bien dar paso a tantas entradas como conexiones haya definidas. Estas entradas son, en último término, las que realmente nos interesarán.

> **NOTA**
>
> *La lista de conexiones mostrada por el Explorador de datos es el reflejo de la información almacenada en un archivo de definiciones global, generado durante la instalación de Delphi. Habitualmente lo encontraremos en* `C:\RAD Studio\FireDAC`.

En esta sección van a describirse los procedimientos a seguir para definir una nueva conexión, explorar conexiones existentes y usarlas para agregar al módulo de datos los componentes de conexión necesarios.

3.2.1 Definición de conexiones

Salvo que vayamos a utilizar alguna de las conexiones asociadas a bases de datos de ejemplo para realizar alguna prueba, normalmente lo primero que haremos será definir una nueva conexión a fin de poder acceder a la información asociada a nuestro proyecto. Los pasos a seguir para ello son los indicados a continuación:

1. Seleccionamos el nodo correspondiente al controlador FireDAC apropiado a fin de que la nueva conexión tome sus parámetros por defecto.
2. Abrimos el menú contextual de dicho nodo y elegimos la única opción que hay disponible (véase la Figura 3.1).

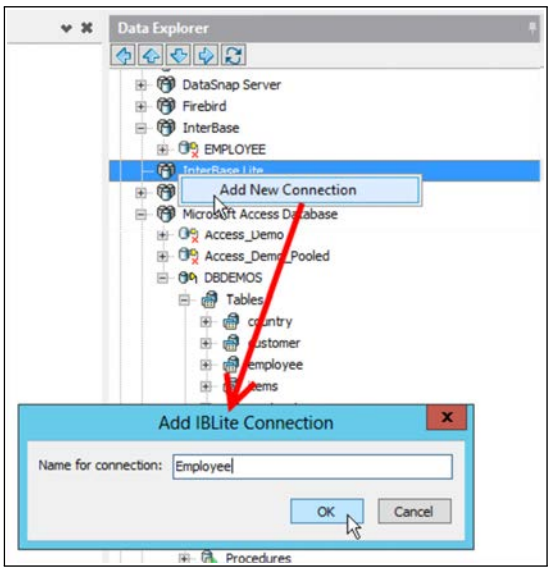


Figura 3.1 AGREGAMOS UNA NUEVA CONEXIÓN ASOCIADA AL CONTROLADOR DE INTERBASE

3. En el pequeño cuadro de diálogo que se abre a continuación debemos introducir el nombre que queremos asignarle a la nueva conexión. No pueden existir dos conexiones con el mismo nombre, por lo que este ha de ser único.
4. Hacemos clic en **OK** para acceder a la ventana **FIREDAC CONNECTION EDITOR**. Esta es la misma que usamos en el capítulo previo. Dependiendo del tipo de controlador seleccionado, la lista de parámetros de la lista inferior cambiará.

Además de los parámetros de conexión, tales como la ruta a la base de datos, el nombre de usuario y contraseña si son necesarios o la codificación de carac-

teres a usar, en la página **OPTIONS** de esta ventana, mostrada en la Figura 3.2, encontramos una extensa lista de opciones de configuración. Con ellas es posible establecer conversiones de tipos de datos entre la base de datos y la aplicación, determinar la manera en que se tratarán las cadenas de caracteres, cómo se gestionarán los bloqueos o las relaciones maestro/detalle, etc.

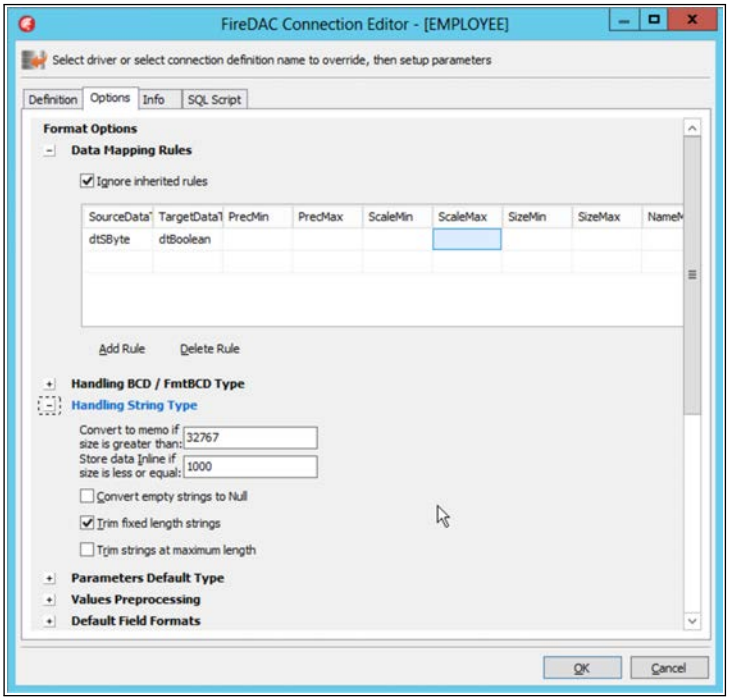


Figura 3.2 OPCIONES DE CONFIGURACIÓN DE LA CONEXIÓN

Una vez que hayamos introducido toda la información necesaria podemos cerrar la ventana, guardando los parámetros de conexión en el archivo global anteriormente citado. Antes, si lo deseamos, podemos usar el botón **TEST** de la página **DEFINITION** para comprobar si la conexión es posible.

Mediante las opciones del menú contextual de una conexión ya existente podemos cambiar su nombre, acceder a esta misma ventana para cambiar su configuración o eliminarla.

3.2.2 Exploración de datos

Disponiendo de una conexión ya definida, podemos usar el Explorador de datos para aquello que denota su nombre: explorar la estructura de la base de datos. Al hacer clic en el botón + que hay a la izquierda de la conexión se procederá a conectar, solicitándonos las credenciales si fuese necesario. En caso de que se trate de un RDBMS remoto, ejecutándose en una máquina distinta a la de desarrollo, lógicamente deberemos contar con conectividad y privilegios de acceso al servidor.

Los nodos de primer nivel de una conexión corresponden a categorías de objetos que pueden existir en la base de datos: tablas, vistas, índices, procedimientos almacenados, etc. Según el tipo de objeto tendremos a nuestra disposicióńn unas opciones u otras. Las tablas y vistas cuentan con la opción VIEW (véase la Figura 3.3), cuya finalidad es mostrarnos los datos que contienen. Además también podemos examinar la lista de columnas de las tablas, los índices que tienen asociados, las claves primarias y foráneas, etc.

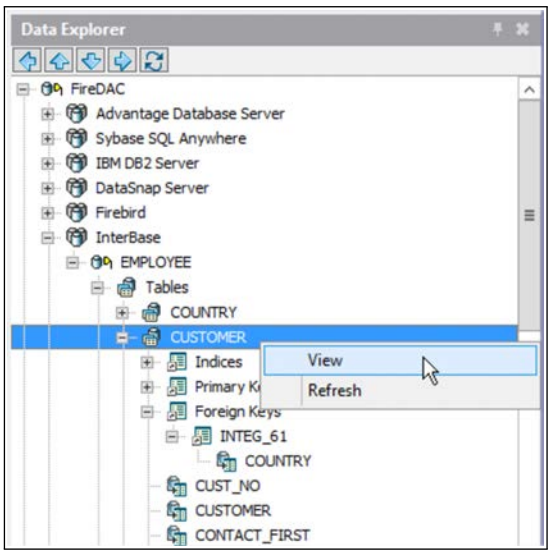


Figura 3.3 EXPLORACIÓN DE LA ESTRUCTURA DE UNA BASE DE DATOS

El uso interactivo de la base de datos que nos permite llevar a cabo el Explorador de datos es sencillamente un atajo, gracias al cual podemos saber qué parámetros precisa un procedimiento almacenado o cómo se llaman las columnas de una tabla sin necesidad de recurrir a la interfaz de usuario propia de cada RDBMS o base de datos. No tenemos, por ejemplo, que abrir el **SQL SERVER MANAGEMENT STUDIO** para realizar dichas tareas, todo el trabajo se desempeña desde el propio entorno de Delphi.

3.2.3 Inserción de componentes de conexión

Una de las tareas iniciales a la hora de desarrollar un proyecto con conexión a datos es la inserción, en el módulo de datos, de los componentes de conexión y los componentes de consulta para recuperar la información. Esta es una tarea en la que el Explorador de datos puede ahorrarnos un trabajo considerable, evitando que todo el trabajo de configuración que hemos hecho al definir la conexión tengamos que repetirlo nuevamente al configurar un `TFDConnection`.

Tomando cualquier tabla o vista desde el Explorador de datos y arrastrándola hasta el módulo de datos[1] conseguiremos que se inserten automáticamente un `TFDConnection`[2] y un `TFDQuery`, ambos apropiadamnete configurados para acceder a los datos de la tabla o vista. También podemos arrastrar y soltar un procedimiento almacenado, en cuyo caso el segundo componente agregado sería un `TFDStoredProc` en lugar de un `TFDQuery`.

Al configurar un componente `TFDConnection` manualmente, tal y como se describió en el capítulo previo, los parámetros que determinan el controlador FireDAC a usar, la localización de la base de datos, etc., se almacenan en el propio componente. Esto implica que al compilar el proyecto estamos introduciendo en el ejecutable toda esa información, por lo que no será posible modificarla salvo que introduzcamos en la aplicación alguna función con tal fin.

Cuando el componente `TFDConnection` se ha añadido mediante arrastrar y soltar, como se ha hecho en la Figura 3.4, la configuración es distinta. Los datos de conexión no están en la propiedad `Params` del componente, sino en el archivo de parámetros FireDAC global. Es la propiedad `ConnectionDefName`

[1] También se podría arrastrar directamente a un formulario.

[2] Si ya existiese un componente de conexión a esa base de datos sería reutilizado, en lugar de agregar otro nuevo.

de `TFDConnection` la que contiene el nombre de la conexión predefinida a usar, de forma que durante la ejecución se buscará el citado archivo de configuración, se localizará en él la conexión indicada (de ahí que su nombre deba ser único) y se cargarán los parámetros de configuración. Esto permite que dichos parámetros sea modificados externamente a la aplicación, simplemente editando el archivo `FDConnectionDefs.ini` correspondiente.

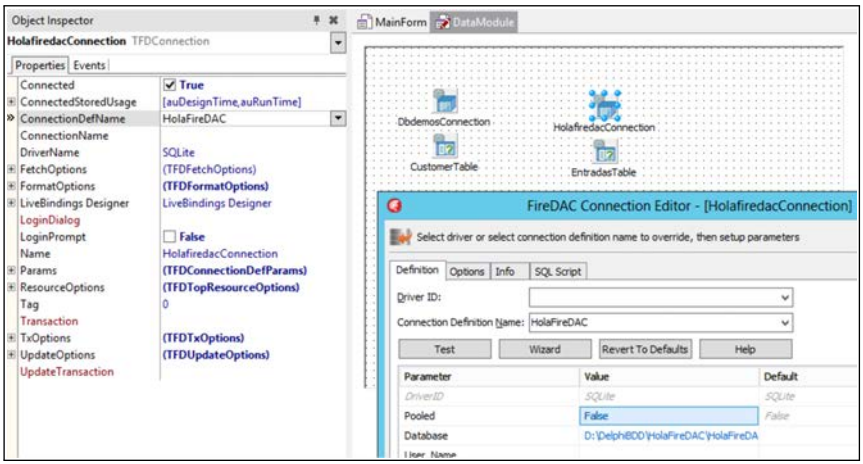


Figura 3.4 COMPONENTES AGREGADOS MEDIANTE ARRASTRAR Y SOLTAR

> **NOTA**
>
> *Lógicamente el archivo global de configuración de FireDAC, instalado junto a Delphi y gestionado por el Explorador de datos, no estará disponible en los equipos en los que vaya a ponerse en explotación la aplicación.*

En la sección siguiente conoceremos una herramienta específica cuya finalidad es editar el contenido de archivos de definición de conexiones FireDAC, ya sea el de configuración global o módulos de definición asociados a proyectos concretos.

3.3 FireDAC Explorer

En el menú **TOOLS** de Delphi encontraremos una opción que abre el programa **FIREDAC EXPLORER**. Esta utilidad es una herramienta genérica para la manipulación de parámetros de conexión de FireDAC almacenados en archivos de definición de conexiones. Por defecto, cuando iniciamos el programa, se abre el archivo de definiciones global de FireDAC. Usando los dos primeros botones que hay en la barra de herramientas (véase la Figura 3.5), sin embargo, podemos tanto crear nuevos archivos como abrir otros, según nuestras necesidades.

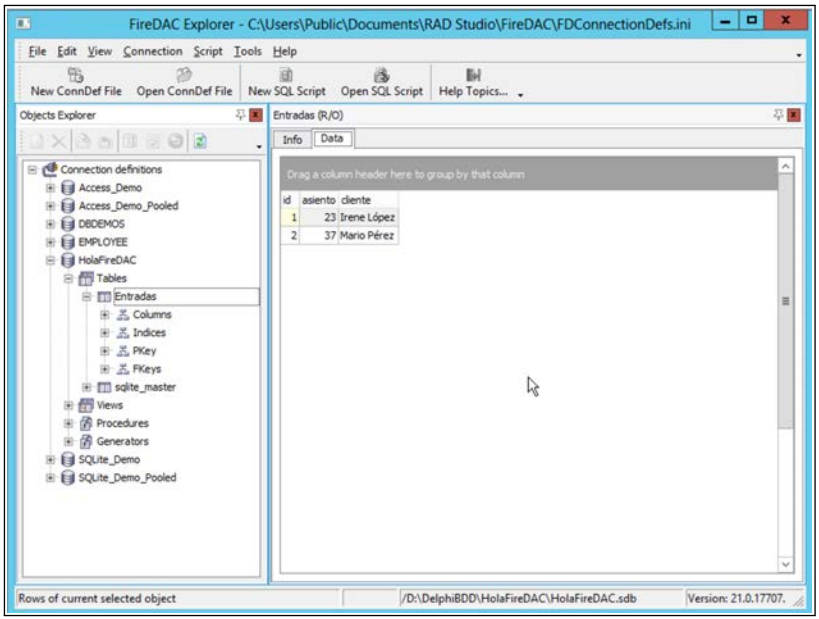


Figura 3.5 FIREDAC EXPLORER

El árbol mostrado en el panel izquierdo se asemeja al contenido del Explorador de datos, ya que contiene una lista de controladores FireDAC y conexiones asociadas a los mismos. Con el menú de opciones contextual podemos definir nuevas conexiones, así como manipular las ya existentes. Abriendo una conexión y seleccionando los objetos que contiene la base de datos, por ejemplo tablas y

vistas, obtendremos en el panel derecho información sobre su estructura, así como acceso a su contenido. Este panel cuenta con funciones que nos permiten agrupar los datos, filtrarlos, ordenarlos, etc. En la Figura 3.6, por ejemplo, se han agrupado los pedidos de una base de datos por el código de cliente a que pertenecen.

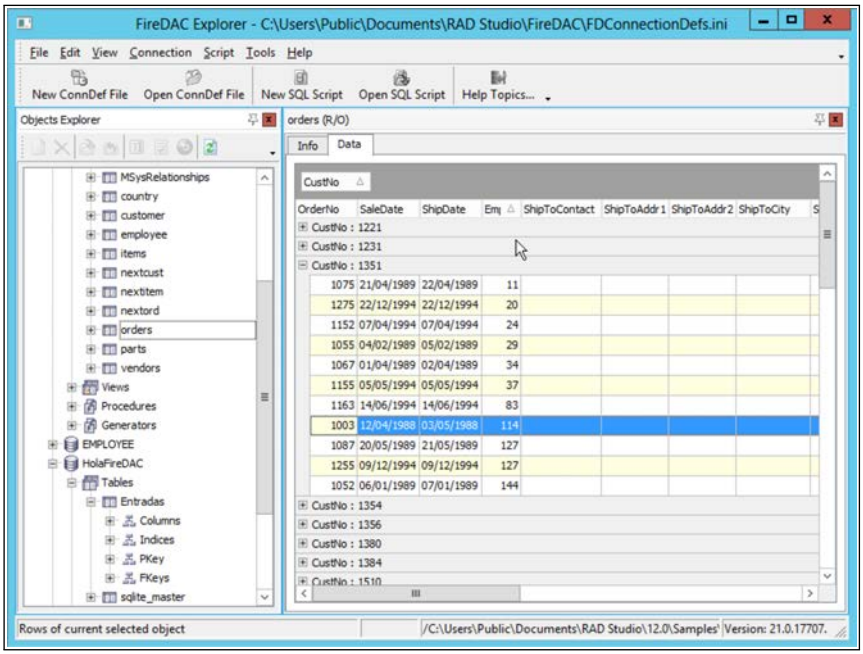


Figura 3.6 ES POSIBLE CREAR GRUPOS, ORDENAR Y FILTRAR LOS DATOS

Asimismo podemos crear *scripts* SQL, con la opción **NEW SQL SCRIPT**, preparando y probando las consultas que posteriormente utilizaríamos en el proyecto. Al crear un nuevo guión SQL se abrirá un panel de edición simple, conectado a la base de datos que tuviésemos seleccionada en el panel izquierdo, permitiéndonos operar de manera interactiva. Estos guiones SQL podemos guardarlos y, posteriormente, recuperarlos para su uso, por ejemplo en un componente `TFDQuery`.

NOTA

Si tenemos en el sistema alias BDE de proyectos anteriores, desde FireDAC Explorer podemos importar esas definiciones a fin de acceder a las mismas bases de datos usando componentes FireDAC. Para ello usaríamos la opción **CONNECTION—IMPORT BDE ALIASES***, seleccionando a continuación los alias BDE que deseemos importar.*

3.3.1 Archivos de definición de conexiones locales a un proyecto

Como se apuntaba anteriormente, el archivo en el que se alojan las definiciones de conexión FireDAC en nuestro equipo de desarrollo no estará disponible en los equipos en los que, finalmente, se despliegue la aplicación. Si los componentes `TFDConnection` usados en esta apuntan, mediante su propiedad `ConnectionDefName`, a dichas definiciones el programa no podrá funcionar al no encontrarlas.

La solución es simple: incorporar al proyecto un archivo de definición de conexiones local, a medida para la aplicación. Los pasos a seguir para ello son los siguientes:

1. Mediante la opción **TOOLS—FIREDAC EXPLORER** del menú principal de Delphi abrimos el FireDAC Explorer.
2. Hacemos clic en el botón **NEW CONNDEF FILE** y seleccionamos la carpeta en la que se alojará el nuevo archivo. Dicha carpeta sería la que vaya a alojar el ejecutable de nuestro proyecto.
3. Usando las opciones del panel izquierdo creamos las conexiones que necesitemos y comprobamos su funcionamiento. En las rutas a bases de datos deberíamos utilizar siempre caminos relativos, no absolutos que hagan referencia a las unidades y estructura de directorios de nuestro equipo de desarrollo.
4. Cerramos el FireDAC Explorer. A partir de ahora, al ejecutar la aplicación esta encontrará el archivo local de definiciones y usará su contenido.

Siempre que incluyamos el archivo de definición de conexiones en el proceso de despliegue, situándolo en la carpeta donde esté el ejecutable, y la propiedad `ConnectionDefName` del componente `TFDConnection` coincida con el nombre de las definiciones alojadas en dicho archivo, nuestra aplicación podrá conectar con las bases de datos que precisa.

> **NOTA**
>
> *Por convención, el archivo de definición de conexiones FireDAC siempre tendrá por nombre* `FDConnectionDefs.ini`.

3.3.2 Estructura de un archivo de definición de conexiones FireDAC

Si al desplegar la aplicación necesitásemos efectuar cambios en la definición de alguna conexión, y no dispusiésemos de la anterior herramienta, podemos recurrir a cualquier editor de texto básico. Al fin y al cabo no es más que un archivo de texto con una estructura muy básica, como la de cualquier archivo de tipo `INI`.

Cada entrada viene denotada por una cabecera `[NombreConexión]`. Tras ella, cada línea contendrá una pareja `clave=valor` estableciendo los distintos parámetros de la conexión.

En el listado siguiente se muestra el contenido de un archivo de definición FireDAC con dos conexiones. La primera es una base de datos InterBase de ejemplo, mientras que la segunda corresponde a la base de datos creada como ejercicio en el capítulo previo. Observa cómo el parámetro `Database` de esta última usa una ruta relativa para hacer referencia a la base de datos.

```
1 [FDConnectionDefs.ini]
2 Encoding=UTF8
3
4 [EMPLOYEE]
5 DriverID=IB
6 Protocol=TCPIP
```

```
7  Database=localhost:C:\ProgramData\Embarcadero\InterBase\↙
       ↳ gds_db\examples\database\employee.gdb
8  User_Name=sysdba
9  Password=masterkey
10 CharacterSet=
11 ExtendedMetadata=True
12
13 [HolaFireDAC]
14 Database=..\HolaFireDAC\HolaFireDAC.sdb
15 DriverID=SQLite
```

Listado 3.1 Archivo `FDConnectionDefs.ini`

3.4 FireDAC Monitor

En el menú **TOOLS** encontramos una segunda opción, con el título **FIREDAC MONITOR**, que da acceso a otra de las herramientas de FireDAC. El objetivo de esta es inspeccionar toda la comunicación entre una aplicación que utiliza componentes FireDAC y la base de datos. Se trata, por tanto, de una utilidad de asistencia tanto a la depuración como al análisis del rendimiento.

Realmente podemos generar un registro de toda la actividad de los componentes FireDAC usados en la aplicación sin necesidad de usar la anterior herramienta, dirigiendo la salida a un archivo de registro (*log*) o bien con una configuración a medida. Con el FireDAC Monitor, no obstante, ya tenemos lo necesario para inspeccionar dicha actividad de forma interactiva.

3.4.1 Componentes de supervisión FireDAC

En la Paleta de herramientas, concretamente en la página **FIREDAC LINKS**, encontrarás tres componentes relacionados con la monitorización de FireDAC. Cada uno de ellos envía los eventos registrados a un destino distinto:

- `TFDMoniCustomClientLink`: Envía el flujo de eventos a un gestor que habríamos de implementar nosotros, respondiendo al evento `OnOutput`.

En el método asociado a dicho evento procesaríamos la información de registro como nos conviniese, lo cual nos permite un tratamiento totalmente personalizado.

- `TFDMoniFlatFileClientLink`: Almacena la información de los eventos generados por FireDAC en un archivo de texto simple. Al finalizar el proceso el componente muestra la lista de archivos de traza.
- `TFDMoniRemoteClientLink`: Transmite toda la información de los eventos producidos por FireDAC a la herramienta FireDAC Monitor. Esta debe estar en ejecución antes de comenzar el proceso de trazado.

Cada uno de estos tres componentes cuenta con un conjunto de propiedades que facilita su configuración, por ejemplo determinando el puerto TCP/IP en el que un `TFDMoniRemoteClientLink` buscará la utilidad FireDAC Monitor, o estableciendo la ruta y el nombre del archivo en el que el componente `TFDMoniFlatFileClientLink` escribirá el registro.

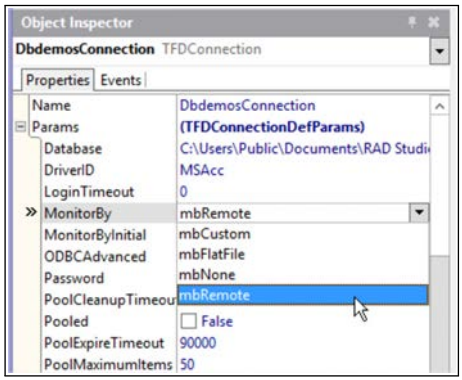


Figura 3.7 CONFIGURACIÓN DE SUPERVISIÓN EN EL COMPONENTE `TFDCONNECTION`

Además de incluir el componente apropiado en nuestro módulo de datos, también tendremos que especificar en el componente `TFDConnection` el tipo de supervisión que se desea utilizar[3]. Para ello tendremos que modificar la

[3]En un mismo módulo de datos podemos tener varios `TFDConnection` y múltiples componentes de supervisión, estableciéndose de esta forma qué método usará cada conexión.

propiedad MonitorBy, seleccionando uno de los cuatro valores disponibles (véase la Figura 3.7). Finalizada esta configuración estaríamos en disposición de comenzar a inspeccionar la actividad de FireDAC durante la ejecución del programa.

3.4.2 La interfaz de FireDAC Monitor

Esta utilidad tiene una interfaz de usuario extremadamente sencilla. Como puede apreciarse en la Figura 3.8, la parte superior está ocupada por una barra de herramientas que facilita las operaciones más comunes: borrar la lista de eventos, guardarla, pausar/reanudar el registro de eventos y mantener la ventana en primer plano.

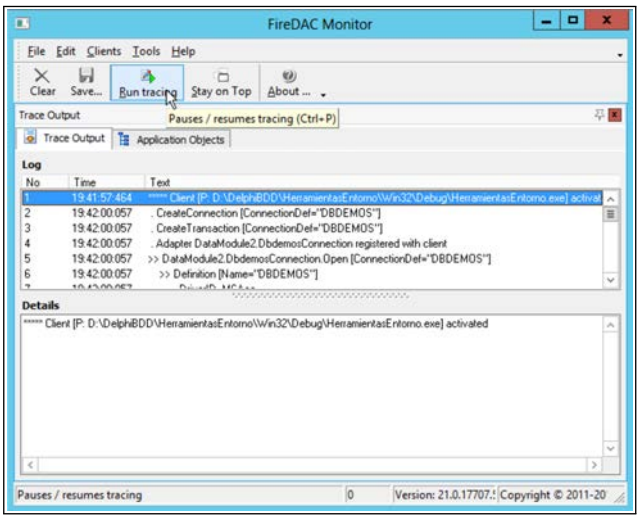


Figura 3.8 INTERFAZ DE USUARIO DEL PROGRAMA FIREDAC MONITOR

La parte central de la interfaz cuenta con dos paneles. El superior muestra una lista con cada uno de los eventos recibidos de la aplicación que está supervisándose, incluyendo una marca de tiempo y un resumen abreviado. En el panel inferior se facilitan detalles sobre el evento seleccionado en la lista anterior.

3.5 Herramientas asociadas a componentes FireDAC

Como ya sabemos, el componente `TFDConnection` tiene asociado un editor de configuración específico, accesible desde su menú contextual o simplemente haciendo doble clic sobre él. Este no es el único componente FireDAC que cuenta con herramientas a medida.

En el menú contextual de un `TFDStoredProc` encontraremos las opciones **EXECUTE** y **NEXT RECORD SET**, en el de un `TFDTable` también se ofrece la opción **FIELDS EDITOR** y en el de un `TFDQuery` se agrega la opción **QUERY EDITOR** (véase la Figura 3.9). Cada una de ellas ejecuta una función o da paso a una utilidad independiente, actuando siempre sobre el conjunto de datos vinculado al componente.

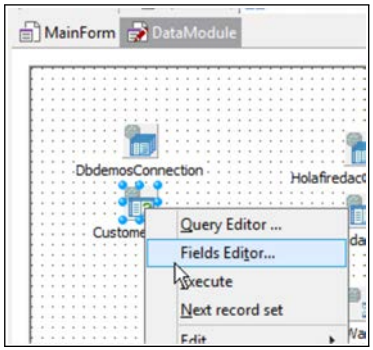


Figura 3.9 OPCIONES ASOCIADAS AL COMPONENTE `TFDQUERY`

3.5.1 El Editor de campos

Los componentes `TFDTable` y `TFDQuery` se utilizan para obtener información de la base de datos. Dicha información está estructurada en columnas, cada una de las cuales tiene un nombre, un tipo de dato y, posiblemente, restricciones y otros atributos.

Si examinamos el contenido de un componente `TFDQuery` en la ventana **STRUCTURE** (en segundo plano en la Figura 3.10), comprobaremos que existen varias ramas inicialmente vacías. Estas se corresponden con las propiedades `Aggregates`, `Constraints`, `Fields`, `Indexes`, `Macros` y `Params` del componente, todas ellas colecciones que almacenan información sobre campos agregados, restricciones, columnas, índices, macros y parámetros, respectivamente. La colección `Fields` tiene asociado un editor específico (en primer plano en la misma Figura 3.10), el Editor de campos, al que da paso la opción **FIELDS EDITOR** antes mencionada.

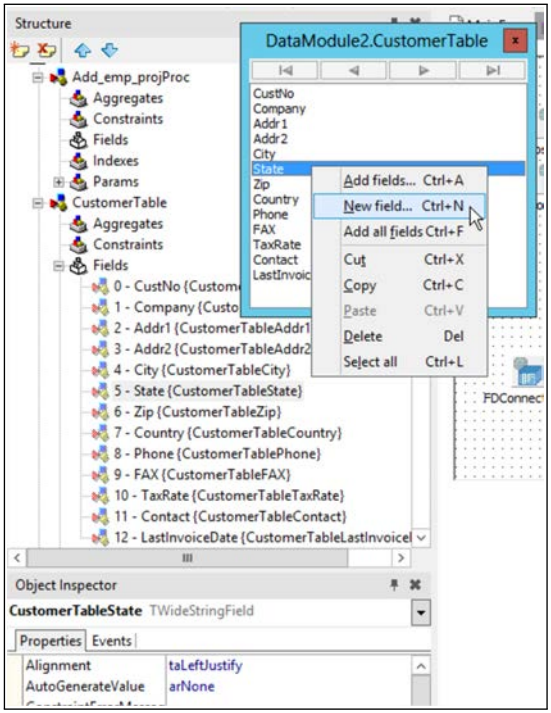


Figura 3.10 EL EDITOR DE CAMPOS Y SU MENÚ CONTEXTUAL

Salvo que establezcamos lo contrario, el contenido de las anteriores colecciones se determinará automáticamente durante la ejecución de la aplicación, una vez que se conecte con la base de datos y el componente `TFDTable` o

`TFDQuery` pueda obtener toda la información de definición de las columnas, índices, etc. Por cada columna se generará un objeto de clase `TField`, configurado a través de sus propiedades y posteriormente almacenado en la colección `Fields`.

Al abrir el Editor de campos lo encontraremos inicialmente vacío. Esto implica que la información de las columnas se obtendrá dinámicamente, durante la ejecución. Mediante la opción **ADD FIELD** del menú contextual accederemos a una ventana con la lista completa de columnas correspondientes a la tabla o consulta, pudiendo seleccionar aquellas que queremos tener disponibles. También podemos usar la opción **ADD ALL FIELDS** para agregarlas todas. Lo que conseguimos con esto es definir el contenido de la colección `Fields` en la fase de diseño, estableciendo la configuración de cada campo de manera estática.

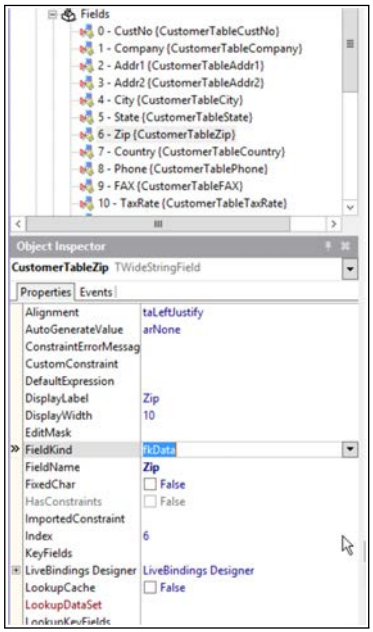


Figura 3.11 PROPIEDADES DE UN OBJETO `TFIELD` EN EL INSPECTOR DE OBJETOS

Usando el Inspector de objetos (véase la Figura 3.11) podemos personalizar la configuración de cada `T Field`, modificando los valores establecidos au-

tomáticamente a partir de la información obtenida de la base de datos. Podríamos, por ejemplo, introducir una expresión en la propiedad `CustomContrainst` para definir una restricción a medida que precise nuestra aplicación.

> **NOTA**
>
> *Al definir en la fase de diseño los campos que ha de tener un conjunto de datos, agregándolos a la colección `Fields`, estamos fijando su estructura respecto a cómo estaba la base de datos durante el desarrollo. Si hubiese algún cambio posterior en ella, la aplicación detectaría la inconsistencia de manera inmediata al ejecutarse.*

Adición de campos calculados

Una de las tareas para las que suele utilizarse el Editor de campos es la definición de campos calculados a un conjunto de datos. Para ello se utiliza la opción **NEW FIELD** de su menú contextual. Esta da paso al cuadro de diálogo mostrado en la Figura 3.12.

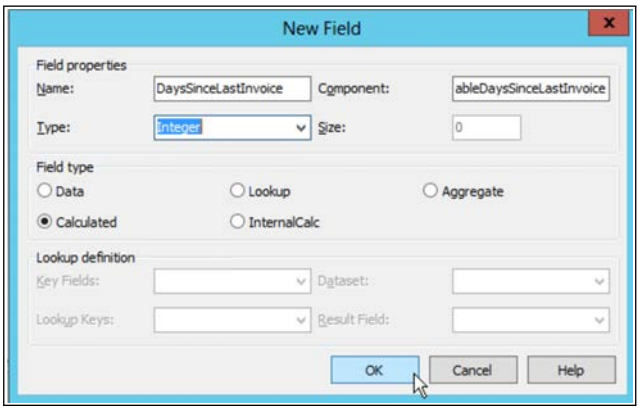


Figura 3.12 DEFINICIÓN DEL NUEVO CAMPO AGREGADO AL CONJUNTO DE DATOS

Además de establecer el nombre que tendrá la nueva columna en el conjunto de datos, así como el nombre del componente, debemos especificar el tipo de

campo eligiendo entre las cinco posibilidades ofrecidas en el apartado **FIELD TYPE**. Con la opción **CALCULATED** indicaríamos que se tratará de una columna calculada, cuyo contenido se establecerá en ejecución en respuesta al evento `OnCalcFields` del componente `TFDTable` o `TFDQuery` que contiene el campo.

3.5.2 El Editor de consultas

Otra de las herramientas relacionadas con la gestión de datos que encontramos en el menú contextual de ciertos componentes FireDAC es el Editor de consultas (véase la Figura 3.13). Como su propio nombre denota, su finalidad es facilitar la composición de consultas SQL, permitiendo probarlas de formar interactiva y pudiendo usar parámetros sustituibles.

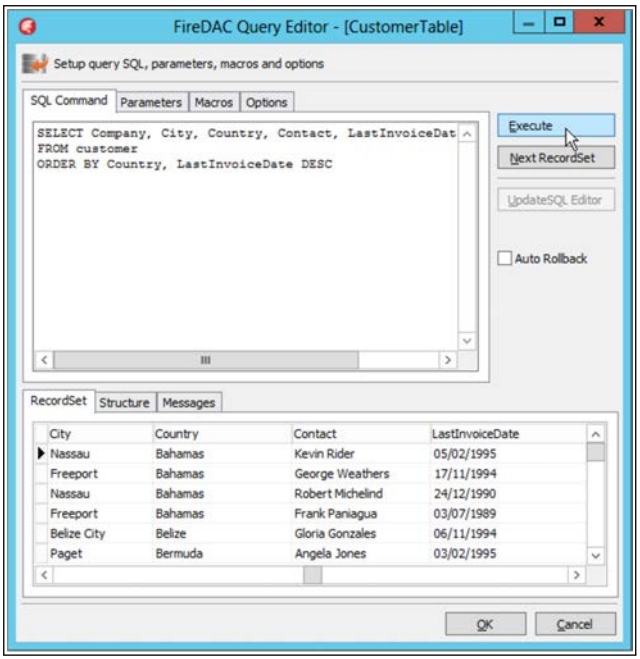


Figura 3.13 EL EDITOR DE CONSULTAS NOS PERMITE PROBAR LAS SENTENCIAS SQL

La página **SQL COMMAND** de esta herramienta será la que utilicemos con más frecuencia. En ella aparecerá la sentencia SQL que está utilizándose actualmente. Si hemos obtenido el componente al arrastrar una tabla desde el Explorador de datos, dicha consulta será tan simple como el clásico (y nada recomendable) `SELECT * FROM tabla`. Modificando la consulta podremos seleccionar las columnas que realmente necesitamos en la aplicación, filtrar las filas, establecer su orden, si fuese necesario agrupar los datos, etc. Cada vez que modifiquemos la consulta no tenemos más que hacer clic en **EXECUTE** para ejecutarla, obteniendo en la parte inferior el conjunto de datos resultante e información sobre su estructura.

En una consulta SQL pueden utilizarse parámetros sustituibles. Estos representan un valor que se usaría en algún punto de la consulta pero que no es conocido en el momento de escribir esta, debiendo facilitarse con posterioridad, normalmente durante la ejecución. La mostrada en el Listado 3.2 es un ejemplo de este tipo de consulta. El parámetro se ha utilizado en la cláusula `WHERE` y se llama `RefDate`. Al colocar dos puntos delante del nombre estamos indicando que se trata de un parámetro sustituible.

```
1 SELECT Company, City, Country, Contact, LastInvoiceDate
2 FROM customer
3 WHERE LastInvoiceDate > :RefDate
4 ORDER BY Country, LastInvoiceDate DESC
```

Listado 3.2 Consulta con parámetros sustituibles

La página **PARAMS** del Editor de consultas nos servirá para configurar los parámetros sustituibles, estableciendo su tipo, tamaño y el valor que le queremos dar para probar la consulta. En la Figura 3.14 se muestra la configuración asociada al parámetro `RefDate` de la consulta de ejemplo previa, indicando que su tipo es `ftDate` y que su valor inicial será `01/01/1995`.

Para facilitar en ejecución el valor que se desea asignar a un parámetro sustituible podemos utilizar la propiedad `Params` del `TFDQuery` o bien recurrir al método `ParamByName`. Ambos miembros los encontraremos documentados en la ayuda electrónica de Delphi. Posteriormente aprenderemos a usar este último en un ejercicio práctico.

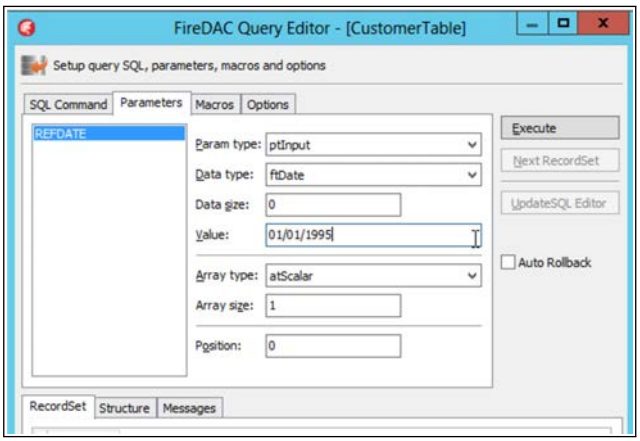


Figura 3.14 CONFIGURACIÓN DE PARÁMETROS USADOS EN LA CONSULTA

3.6 El asistente de Live Bindings

La última herramienta de la que vamos a ocuparnos en este capítulo es el asistente **LIVEBINDINGS WIZARD**. Este se abre mediante el último de los botones que hay en la barra lateral del **LIVEBINDINGS DESIGNER**, el panel de enlace visual entre datos y controles de la interfaz de usuario que abríamos con la opción **BIND VISUALLY** del menú contextual de cualquier control.

Este asistente cuenta con un número variable de pasos, dependiendo de la acción que elijamos en la primera página. Como se aprecia en la Figura 3.15 las opciones disponibles son cinco. La última nos permite crear un nuevo conjunto de datos (un componente `TFDTable` o `TFDQuery`), facilitando la selección de una conexión o un controlador y agregando al contenedor actual los componentes necesarios. Las otras cuatro opciones tienen por finalidad c onectar un control de interfaz de usuario con un campo de un conjunto de datos o una de las propiedades de un componente.

El procedimiento para conectar un control a un campo de la base de datos, o una cuadrícula con un conjunto de campos, es fundamentalmente el mismo. En el primer caso tendríamos que seleccionar el tipo de control que queremos utilizar para mostrar la información del campo, mientras que en el segundo se usaría un

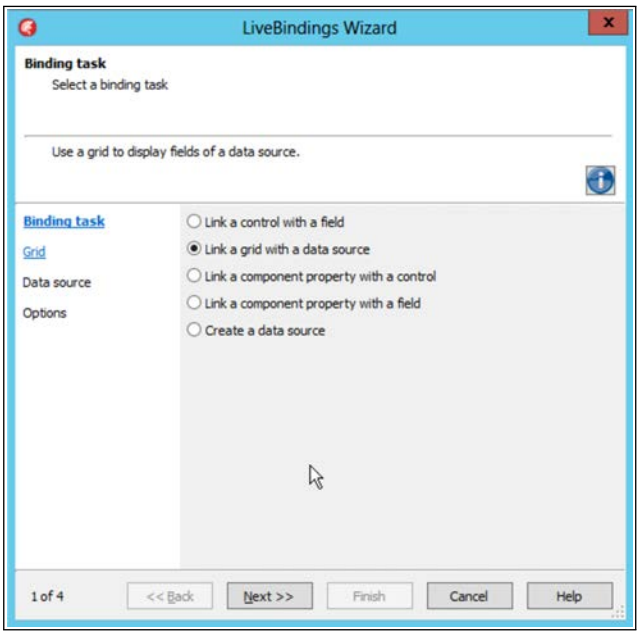


Figura 3.15 ASISTENTE PARA LIVE BINDINGS

componente `TGrid` o equivalente. A partir de aquí podríamos seleccionar uno de los conjuntos de datos ya existentes, por ejemplo agregados antes al módulo de datos, o bien optar porque el asistente los genere también.

Al finalizar el asistente tendríamos en el formulario todos los componentes y controles necesarios, con los vínculos entre ellos adecuadamente establecidos lo cual nos ahorra una cantidad considerable de tiempo.

3.7 En la práctica

Terminamos poniendo en práctica el uso de muchas de las herramientas descritas en las secciones de este capítulo. Para ello desarrollaremos una sencilla aplicación cuya finalidad será mostrar datos de una tabla de una de las bases de datos

de ejemplo distribuidas con Delphi, facilitando también su edición. Además el programa permitirá establecer un filtro a la información mostrada, añadiendo también una columna cuyo contenido será obtenido mediante un cálculo.

> **EJEMPLO 3.1** `HerramientasEntorno.dproj`
>
> Puedes encontrar este proyecto completo en la carpeta `HerramientasEntorno`. Para usarlo en tu equipo tendrás que editar el archivo de definición de conexiones, ajustando la ruta a la base de datos, y a continuación tendrás que copiar dicho archivo a la carpeta que contiene el ejecutable.

Comenzamos iniciando un nuevo proyecto vacío de tipo **MULTI-DEVICE APPLICATION** y, como hicimos en el capítulo previo, agregaremos un módulo de datos al mismo. Mediante la opción **FILE—USE UNIT** vincularemos el formulario con el módulo de datos, a fin de poder acceder desde el primero a los componentes de datos del segundo.

3.7.1 Conexión y selección de datos

Usando el Explorador de datos localizamos en el nodo **MICROSOFT ACCESS DATABASE** la base de datos de ejemplo `DBDEMOS`. Esta contiene una tabla llamada `customer`. Hacemos clic sobre ella y la arrastramos hasta soltarla sobre el módulo de datos. Esta acción agregará al módulo de datos dos componentes: un `TFDConnection` y un `TFDQuery`. El primero tendrá en su propiedad `ConnectionDefName` el valor `DBDEMOS`, apuntando a la definición de conexión almacenada en el archivo de configuración global de FireDAC. El segundo contendrá una consulta de selección para obtener todas las columnas y filas de la tabla `customer`.

El único componente FireDAC adicional que necesita la aplicación sería un `TFDGUIxWaitCursor`. Lo localizamos, en la Paleta de componentes o usando el recuadro de búsqueda de **IDE INSIGHT**, y lo insertamos también en el módulo de datos.

NOTA

Al agregar los componentes con arrastrar y soltar desde el Explorador de datos, en lugar de insertarlos y configurarlos manualmente como hicimos en el capítulo previo, no es necesario añadir el componente `TFDPhysMSAccessDriverLink` *ya que se agregan las referencias a los módulos necesarios automáticamente.*

3.7.2 Configuración para la inspección de actividad

Queremos examinar la actividad de la aplicación mientras operamos con ella, por lo que vamos a añadir al módulo de datos un componente `TFDMoniRemoteClientLink`. Mediante sus propiedades `Host` y `Port` podemos configurar la dirección del equipo y el puerto TCP/IP en que está a la escucha la utilidad FireDAC Monitor[4]. Nos aseguramos de dar el valor `True` a su propiedad `Tracing`.

A continuación seleccionamos el componente `DbdemosConnection` agregado antes, vamos al Inspector de objetos, desplegamos la propiedad `Params`, abrimos la lista asociada a la subpropiedad `MonitorBy` y elegimos la opción `mbRemote`. Esto es todo lo que necesitamos hacer para poder inspeccionar la actividad de la aplicación.

3.7.3 Edición de la consulta de selección de datos

Abrimos el menú contextual asociado al componente `CustomerTable`, cuyo tipo es `TFDQuery`, y seleccionamos la opción **QUERY EDITOR**. Usamos el Editor de consultas para modificar la consulta de selección, introduciendo la que

[4]Podemos configurar el puerto de escucha de este programa mediante el campo **PORT** de la lista de parámetros de configuración, a la que se accede con la opción **TOOLS—TRACE OPTIONS**.

se mostró anteriormente en el Listado 3.2. Con ella seleccionamos solamente cinco columnas de la tabla, filtramos las filas según un parámetro sustituible y fijamos el orden de las filas.

A continuación, aún en el Editor de consultas, abrimos la página **PARAMS** y configuramos el parámetro `RefDate`, estableciéndolo como parámetro de entrada, de tipo `ftDate` y con el valor inicial que deseemos. Ahora podemos cerrar el Editor de consultas.

Con el objetivo de configurar los campos de forma estática, en la fase de diseño, abrimos el Editor de campos del mismo componente `TFDQuery` y elegimos la opción **ADD ALL FIELDS**. Comprobaremos que los únicos campos que aparecen son los cinco referenciados en la cláusula `SELECT` de la consulta.

3.7.4 Adición de un campo calculado

Además de los datos que recuperamos de la tabla, seleccionados en la anterior consulta, también queremos que el programa muestre un dato adicional: el número de días transcurridos desde la última factura de cada cliente. Para ello agregaremos al `TFDQuery` un campo calculado. Los pasos a seguir para ello son los indicados a continuación:

1. Abrimos el menú contextual del Editor de campos y elegimos la opción **NEW FIELD**.
2. Se abre la ventana **NEW FIELD**, en la que comenzamos por introducir el valor `DaysSinceLastInvoice` como nombre del campo.
3. Seleccionamos `Integer` como tipo del nuevo campo y marcamos la opción **CALCULATED**, tal y como se mostró en la Figura 3.12. Hacemos clic en **OK** para cerrar el cuadro de diálogo.
4. Seleccionamos el `TFDQuery` y, usando la página **EVENTS** del Inspector de objetos, localizamos el evento `OnCalcFields`, haciendo doble clic sobre él.
5. Introducimos en el método asociado a dicho evento el código mostrado en el Listado 3.3, con el cual se calcula el valor que habría de tomar el nuevo campo.

```
1 procedure TDataModule2.CustomerTableCalcFields(DataSet: ↙
    ↳ TDataSet);
2 begin
3   CustomerTableDaysSinceLastInvoice.Value :=
4         DaysBetween(System.SysUtils.Date, ↙
            ↳ CustomerTableLastInvoiceDate.AsDateTime);
5 end;
```

Listado 3.3 Expresión para obtener el valor del campo calculado

`CustomerTableDaysSinceLastInvoice` es el nombre que se ha dado a la variable que representa al campo calculado, a partir del nombre que nosotros introdujimos antes y usando como prefijo el nombre del componente `TFDQuery`. Mediante su propiedad `Value` le asignamos el valor que debe tomar, calculando la diferencia en días[5] entre la fecha actual, obtenida con `Date`, y la fecha de la última factura. Esta se almacena en un formato de fecha propia de la base de datos, por lo que usamos la propiedad `AsDateTime` para convertirla.

3.7.5 Diseño de la interfaz de usuario

Ya tenemos en nuestro módulo de datos todo lo que necesitamos. Ahora llega el momento de diseñar la interfaz de usuario de la aplicación. Volvemos al formulario, que estaba vacío, e introducimos los siguientes elementos:

- `TPanel`: Este componente nos servirá como contenedor para los dos controles siguientes. Asignamos el valor `Top` a su propiedad `Align` para colocarlo en la parte superior del formulario y, si es necesario, ajustamos su altura.
- `TCheckBox`: Su finalidad será permitir activar y desactivar la conexión entre la interfaz y el conjunto de datos que va a mostrarse. Modificaremos su propiedad `Text` asignándole el valor `'Activar conexión'`.
- `TDateEdit`: Servirá para introducir una fecha que nos permitirá filtrar las filas mostradas, obteniendo únicamente aquellas que cuentan con facturas

[5] Para poder usar la función `DaysBetween` tendremos que agregar el módulo `DateUtils` a la cláusula `uses` del módulo de datos.

a partir de la fecha indicada. Opcionalmente podemos colocar un control `TLabel` delante para especificar su finalidad.

- `TGrid`: Este control será el que muestre la información procedente de la base de datos. Modificaremos su propiedad `Align` asignándole el valor `Client`, de forma que ocupe todo el espacio disponible salvo el usado por el `TPanel`.

El aspecto del formulario, una vez insertados los componentes indicados, sería similar al que se muestra en la Figura 3.16. El `TGrid` no muestra aún información dado que no hemos establecido la conexión con los componentes de datos.

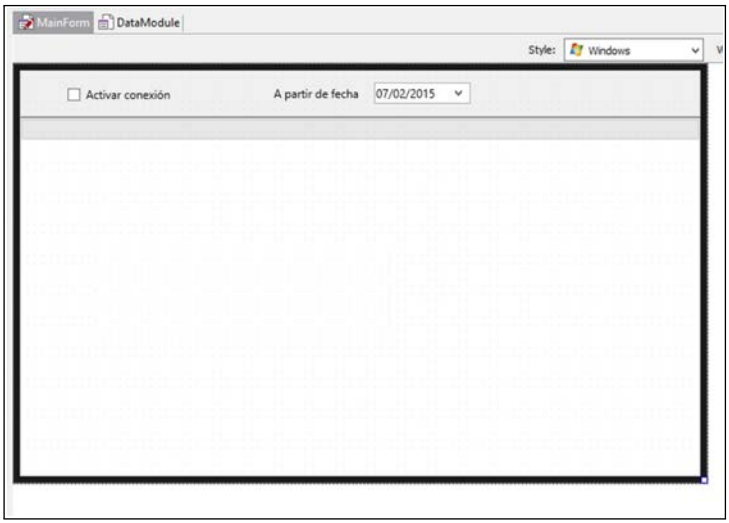


Figura 3.16 FORMULARIO FINALIZADA LA INSERCIÓN DE COMPONENTES

3.7.6 Conexión entre interfaz y datos

El paso siguiente consiste en vincular la interfaz de usuario, en este caso el componente `TGrid`, con el conjunto de datos en el que reside la información sobre

la que se trabajará. Para esta tarea recurriremos al asistente de Live Bindings que se describió anteriormente. El procedimiento a completar es el descrito a continuación:

1. Usamos la opción **BIND VISUALLY** del `TGrid` para abrir el panel **LIVE-BINDINGS DESIGNER** y hacemos clic en el botón que pone en marcha el asistente.

2. Seleccionamos la opción **LINK A GRID WITH A DATA SOURCE** en la primera página y hacemos clic en el botón **NEXT**.

3. En el segundo paso tenemos dos opciones: seleccionar un componente cuadrícula ya existente en la interfaz de usuario o bien crear una nueva y agregarla al formulario. Usamos la primera, eligiendo el componente `TGrid` que aparece como único elemento de la lista y de nuevo hacemos clic en **NEXT**.

4. A continuación tenemos que seleccionar el conjunto de datos que queremos enlazar con la cuadrícula anterior eligiendo, como en el caso previo, entre uno existente o creándolo. Nosotros escogeremos el `TFDQuery` que tenemos en el módulo de datos.

5. Finalmente podemos marcar una opción que añadiría a la interfaz un control de navegación. Nos limitamos a hacer clic en el botón **FINISH**.

Al terminar, el asistente introducirá en el formulario dos componentes: un `TBindSourceDB`, vinculado al `TDBQuery`, y un `TBindingList` para mantener la lista de vínculos.

> **NOTA**
>
> *Si ahora damos el valor `True` a la propiedad `Active` del `TDBQuery`, en el `TGrid` deberíamos ver la información obtenida de la base de datos. Dejaremos dicha propiedad con el valor `False` que tiene por defecto.*

3.7.7 Código de activación y filtrado

Solo resta un pequeño detalle para finalizar el desarrollo de la aplicación: gestionar el evento `OnChange` del control `TCheckBox` que determina si la conexión estará activa o no. Será en ese mismo controlador de evento donde establezcamos el filtro para las filas, asignando el valor introducido en el `TDateEdit` al parámetro `RefDate` de la consulta. Hacemos doble clic sobre el `TCheckBox` e introducimos el código siguiente:

```
1 procedure TMyForm.CheckBox1Change(Sender: TObject);
2 begin
3   with MyDataModule.CustomerTable do
4   begin
5     ParamByName('RefDate').AsDate := DateEdit1.Date;
6     Active := CheckBox1.IsChecked;
7   end;
8 end;
```

Listado 3.4 Código asociado al evento `OnChange` del `TCheckBox`

3.7.8 Probando la aplicación

Antes de lanzar la ejecución del proyecto, para lo cual basta con pulsar **F9**, abriremos el FireDAC Monitor. Esto permitirá que nuestro programa pueda contactar con dicha aplicación, a fin de examinar toda la actividad relativa al acceso a la base de datos.

Al ejecutar el programa el formulario aparecerá vacío. Incluso si activamos la conexión es probable que continúe vacío, ya que la fecha de filtrado por defecto será la actual. No tenemos más que desactivar la conexión, establecer una fecha previa y volver a activarla. El resultado debería ser similar al que se muestra en la Figura 3.17, aunque lógicamente los datos dependerán de la fecha que hayamos introducido.

Cada vez que activemos la conexión comprobaremos, mediante el FireDAC Monitor, cómo se desencadena un buen número de eventos. Cada uno de ellos contiene la información de las operaciones realizadas. Puedes realizar una prueba

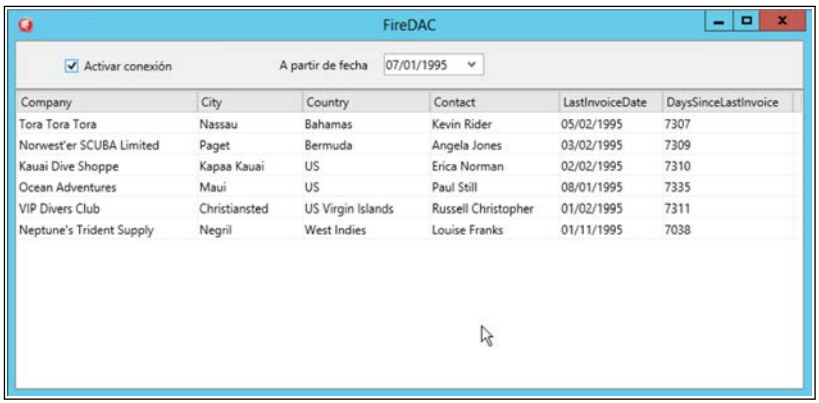


Figura 3.17 EL PROGRAMA EN FUNCIONAMIENTO

muy sencilla: haz clic en la ciudad de una de las filas de datos y modifícala, seleccionando después otra fila para transmitir los cambios. A continuación, en el FireDAC Monitor, localiza la línea en la que se envía el comando `UPDATE CUSTOMER`, como se ha hecho en la Figura 3.18, y haz clic sobre ella para ver los detalles. En el panel inferior podrás ver la sentencia SQL utilizada para efectuar la actualización. En ella se modifica el valor de la columna `CITY` y se toma como referencia (en la cláusula `WHERE`) el número de cliente. De forma similar podríamos inspeccionar cualquier operación de manipulación y transferencia de datos.

3.7.9 Preparar la aplicación para su despliegue

Ejecutar la aplicación en el mismo equipo donde hemos llevado a cabo el desarrollo no implica ningún problema, ya que los parámetros de la conexión están accesibles en el archivo global de configuración de FireDAC y la base de datos se encuentra donde se copió originalmente al instalar Delphi. Si llevásemos el programa a otro ordenador, en el que no estuviese instalado Delphi, al intentar ejecutarlo obtendríamos un mensaje de error.

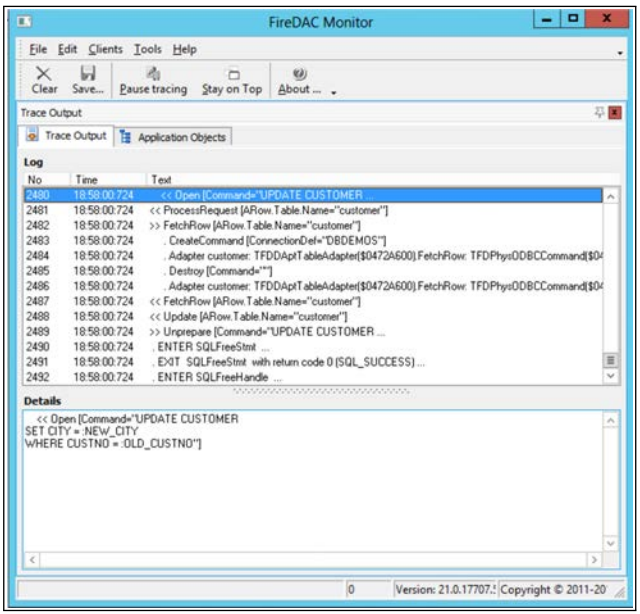


Figura 3.18 INSPECCIONAMOS LA OPERACIÓN DE ACTUALIZACIÓN

Los pasos para preparar nuestro proyecto para su despliegue son muy sencillos en este caso, ya que estamos utilizando una base de datos local. Básicamente tenemos que hacer dos cosas:

- Copiar el archivo `dbdemos.mdb`, en el que está alojada la base de datos, desde su carpeta de origen a la carpeta que contendrá nuestro ejecutable.
- Usar el FireDAC Explorer para crear un nuevo archivo de conexiones conteniendo únicamente la entrada `DBDEMOS`. Esta apuntaría con una ruta relativa al archivo anterior. El archivo `FDConnectionDefs.ini` también acompañaría a nuestro ejecutable.

Teniendo esos elementos, podríamos copiar la carpeta con el ejecutable, la base de datos y el archivo de definición de conexiones a cualquier otro equipo y debería funcionar.

3.8 Resumen

Al finalizar este capítulo ya conocemos la mayor parte de las herramientas de trabajo con bases de datos integradas en Delphi, incluidas utilidades externas como FireDAC Explorer y FireDAC Monitor que nos permitirán realizar la mayor parte de las tareas sin tener que recurrir a herramientas de terceros, todo ello desde el propio entorno de Delphi. En el ejercicio desarrollado en la parte final se ha mostrado cómo usar en la práctica la mayor parte de esas herramientas, creando una nueva aplicación en la que se ha usado una base de datos y una interfaz distinta a la implementada en el capítulo previo.

Con la base adquirida en este y el capítulo anterior, en el próximo nos concentraremos en los detalles del desarrollo de aplicaciones FireMonkey y VCL que usan FireDAC, poniendo especial énfasis en los aspectos relacionados con la interfaz de usuario.

Capítulo 4

INTERFACES DE USUARIO CON CONEXIÓN A DATOS

En el capítulo anterior el objetivo era conocer herramientas del entorno de Delphi relacionadas con el trabajo con bases de datos mediante FireDAC pero de tipo genérico, dejando intencionadamente de lado aquellas dirigidas al desarrollo de la interfaz de usuario de la aplicación. La única excepción fue la introducción del **LIVEBINDINGS DESIGNER**, al tratarse de un elemento fundamental para el seguimiento del ejercicio propuesto.

En contraposición, este capítulo está completamente centrado en el tema de las interfaces de usuario con conexión a orígenes de datos, ya estén basadas en la biblioteca de componentes VCL o en la FMX. Comenzaremos por analizar las

diferencias entre dichas bibliotecas, a fin de que podamos decidir cuál nos conviene en cada caso. Posteriormente se describirán algunos aspectos específicos de los componentes VCL con conexión a datos y, finalmente, se profundizará en el uso de los *LiveBindings*.

4.1 VCL versus FMX

Desde hace años Delphi incorpora dos bibliotecas de componentes con un mismo fin: facilitar el diseño de interfaces de usuario siguiendo el enfoque RAD. La VCL es una vieja conocida para todos aquellos que lleven un tiempo usando Delphi, ya que es la biblioteca que acompaña a Delphi desde su primera versión, habiendo sobrevivido a la multiplataforma CLX que se introdujo con Kylix. Se trata, en consecuencia, de una opción madura y muy probada, en la que se ha trabajado de manera ininterrumpida desde 1995.

La FMX, también conocida como FireMonkey, es una biblioteca de componentes de desarrollo mucho más reciente, pero la inversión hecha en ella y el número de versiones que lleva integrada con Delphi, la hacen también una opción segura para el futuro.

Los apartados de esta sección resumen los aspectos más destacables de estas dos bibliotecas. Nuestro propósito es ayudarte a decidir cuál de las dos elegir en cada caso, según el proyecto cuyo desarrollo vayas a afrontar.

4.1.1 Plataformas objetivo

Una de las diferencias fundamentales entre VCL y FMX, quizá la más importante a la hora de tomar una decisión en cuanto a cuál de las dos elegir, es el conjunto de plataformas objetivo para las que puede compilarse un proyecto que las utilice.

La VCL nació estrechamente ligada a la API de Windows, concretamente a la plataforma que conocemos como **Win32**[1]. Con el tiempo, hace relativamente

[1]En realidad la primera versión de la VCL, incluida en Delphi 1, era de 16 bits. Fue a partir de Delphi 2, tras el lanzamiento de Windows 95, cuando se incluye Win32 como plataforma objetivo.

poco, se añadió también la posibilidad de compilar para **Win64**, aprovechando así la posibilidad de acceder a más memoria. Tras crear un nuevo proyecto basado en la VCL comprobaremos que la única plataforma agregada es Win32. Podemos utilizar la opción **ADD PLATFORM** del menú contextual para añadir otras plataformas. Como se muestra en la Figura 4.1, la única opción adicional disponible es Win64.

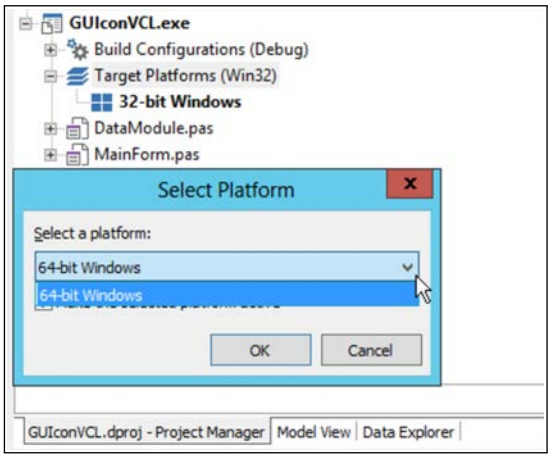


Figura 4.1 VCL SOLO ESTÁ DISPONIBLE PARA WIN32 Y WIN64.

FireMonkey fue adquirida a la empresa KSDev y forma parte de Delphi desde la versión XE2, por lo que no ha sido diseñada para ser compatible con la VCL. Se trata de una biblioteca de componentes multiplataforma, con la que es posible crear proyectos tanto para ordenadores de escritorio con Windows (Win32 y Win64) y Mac OS X como para dispositivos móviles con iOS y Android. De hecho, a partir de Delphi XE7 al iniciar un proyecto de tipo **MULTI-DEVICE** ya encontramos todas las plataformas en el Gestor de proyectos (véase la Figura 4.2), sin necesidad de agregarlas manualmente.

En cuanto a selección de plataformas objetivo, por tanto, la FMX está claramente por delante de la VCL, siendo de hecho la única opción en caso de que necesitemos desarrollar para plataformas móviles.

Figura 4.2 FMX TAMBIÉN ESTÁ DISPONIBLE PARA OS X, IOS Y ANDROID

4.1.2 Hardware y software de base

VCL y FMX fueron desarrolladas en momentos muy diferentes, algo que es apreciable en el aprovechamiento que ambas hacen de los recursos del sistema y que influye asimismo en el hardware y software básico sobre el que pueden ejecutarse.

La FMX es una biblioteca pensada para explotar la potencia de las actuales GPU (*Graphics Processing Units*), un elemento imprescindible en el ordenador o dispositivo donde vayan a ejecutarse las aplicaciones. En el caso de Windows además es preciso contar con una versión reciente de DirectX, la API gráfica que permite a FMX comunicarse con la GPU en lugar de OpenGL que es la API usada en el resto de plataformas.

Prácticamente todos los componentes FMX cuentan con propiedades con las que es posible aplicar transformaciones (`RotationAngle`, `RotationCenter` y `Scale` en la Figura 4.3) y animaciones visuales, facilitando la composición de interfaces de usuario muy atractivas y dinámicas que, además, están preparadas para su funcionamiento en dispositivos con pantallas táctiles. Asimismo es posible crear interfaces gráficas 3D.

En contraposición, la VCL se apoya para hacer su trabajo en la clásica API gráfica de Windows (GDI), en lugar de DirectX u OpenGL. Esto hace que gráficamente sea menos potente, pero a cambio es la única opción si queremos desarrollar aplicaciones para versiones de Windows relativamente antiguas pero todavía muy extendidas, como es el caso de Windows XP.

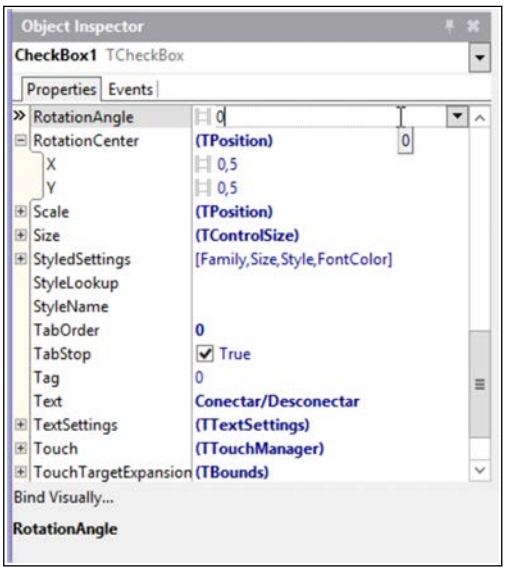


Figura 4.3 PROPIEDADES PARA APLICAR TRANSFORMACIONES EN FMX

NOTA

Los componentes tanto de la VCL como de la FMX, al igual que los de FireDAC según se indicó anteriormente, pasan a formar parte del ejecutable de los proyectos una vez que son compilados, por lo que en ningún caso es necesario redistribuir junto a la aplicación módulos asociados estas bibliotecas.

En consecuencia en la elección de una biblioteca u otra también influirá el hecho de que el proyecto deba o no funcionar en equipos no tan potentes o con versiones anteriores de Windows que, bien sabemos, continúa siendo la plataforma mayoritaria de propósito general en el escritorio.

4.1.3 Compatibilidad hacia atrás

Son muchas las aplicaciones desarrolladas con versiones previas de Delphi que aún siguen en funcionamiento, proyectos que es necesario seguir manteniendo de forma periódica. La mayor parte de ellas están basadas en la VCL, no existiendo una vía de conversión directa de VCL a FMX. Además hay dos razones adicionales para seguir utilizando la VCL en estos casos:

- **Componentes de terceros**: Son multitud los casos en que se utilizaron componentes de terceros, no propios de la VCL pero compatibles con esta, que únicamente están disponibles para dicha biblioteca. Sin romper esa dependencia no es posible convertir el proyecto para usar la FMX, a menos que el desarrollador de esos componentes ofrezca en algún momento una versión FMX de los mismos.

- **Servicios Win32 no en la FMX**: La VCL es una biblioteca específica para Windows, de ahí que ofrezca componentes que aprovechan las características específicas de este sistema, mediante APIs específicas (encontramos muchos de estos componentes, por ejemplo `TTaskbar` y `TJumpList`, en las páginas **WIN32**, **SYSTEM** y **WIN 3.1** de la Paleta de componentes, mostradas en la Figura 4.4). Si se necesitan esos servicios deberemos seguir utilizando la VCL, al menos que en el futuro la FMX también cubra dichas necesidades.

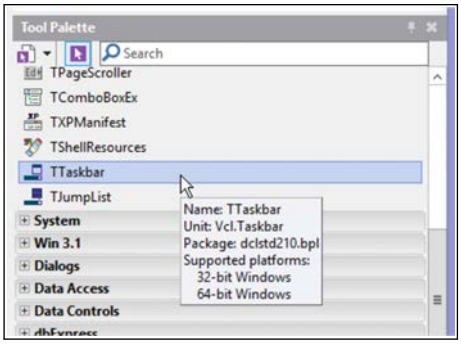


Figura 4.4 COMPONENTES ESPECÍFICOS PARA WINDOWS EN LA VCL

En resumen, elegir entre la VCL y la FMX a la hora de iniciar el desarrollo de un nuevo proyecto con Delphi puede no resultar sencillo, influyendo los aspectos citados y posiblemente algunos más. No obstante, un aspecto que no nos forzará a decantarnos por una u otra es la necesidad de utilizar FireDAC, ya que en ambos casos dispondremos fundamentalmente de las mismas posibilidades.

NOTA

La RTL, biblioteca de servicios básicos de Delphi que opera debajo de la VCL y FMX, es totalmente multiplataforma y compatible con ambas bibliotecas de componentes.

4.2 FireDAC y aplicaciones VCL

Los componentes FireDAC, así como las herramientas que se describieron en el capítulo previo, están también disponibles cuando se trabaja con proyectos basados en la VCL. Vamos a comprobarlo en la práctica creando un simple programa análogo al del ejercicio final del capítulo previo, pero basado en la VCL en lugar de la FMX.

EJEMPLO 4.1 `GUIconVCL.dproj y GUIconFMX.dproj`

En las carpetas `GUIconVCL` y `GUIconFMX` encontrarás dos versiones de un mismo proyecto, el primero utilizando componentes enlazados a datos clásicos de la VCL y el segundo con idéntica funcionalidad pero usando la FMX. En ambos casos existe un formulario principal con una cuadrícula de datos y un formulario secundario en el que se usan otros controles enlazados a datos.

Partimos creando un nuevo proyecto usando la plantilla **VCL FORMS APPLICATION**, tal y como se muestra en la Figura 4.5. El proyecto contará inicial-

mente con un formulario vacío, pero la información de diseño de este se almacenará en un módulo con extensión `.dfm` en lugar de `.fmx`.

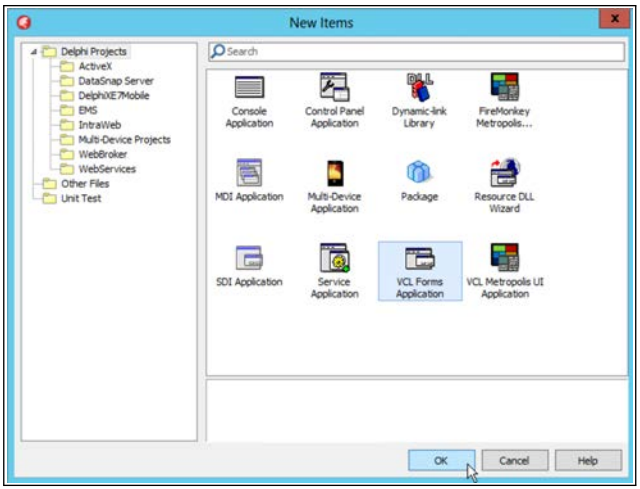


Figura 4.5 CREAMOS UN NUEVO PROYECTO BASADO EN LA VCL

4.2.1 Módulo de datos y componentes FireDAC

Lo primero que haremos será agregar al proyecto un módulo de datos. Este tipo de contenedor es idéntico al que se usa en proyectos FMX, por lo que puede alojar exactamente los mismos tipos de componentes. Acto seguido reproduciremos los pasos indicados a continuación:

1. Localizamos en el Explorador de datos la tabla `customer` de la conexión `DBDEMOS` y la arrastramos hasta el módulo de datos. Esto insertará y configurará los componentes `TFDConnection` y `TFDQuery`.

2. Añadimos el componente `TFDGUIxWaitCursor` y, si queremos, también el `TFDPhysMSAccessDriverLink`, aunque este último no sería necesario al haber creado la conexión desde el Explorador de datos.

3. Finalmente agregamos al módulo de datos un componente `TDataSource`[2], al que llamaremos `CustomerDataSource`, usando su propiedad `DataSet` para conectarlo con el `TFDQuery`, tal y como se aprecia en la Figura 4.6. También daremos el valor `True` a su propiedad `Enabled`.

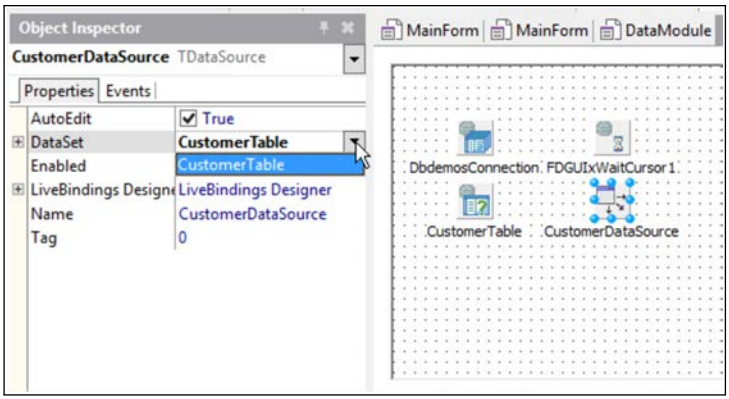


Figura 4.6 AGREGAMOS AL MÓDULO DE DATOS LOS COMPONENTES FIREDAC

A fin de poder hacer referencia a los elementos del módulo de datos desde el formulario, activamos este último en el diseñador y usamos la opción **FILE—USE UNIT** para agregar una referencia al primero.

> **NOTA**
>
> *Al igual que hicimos en el ejercicio del capítulo previo, podríamos usar el Editor de consultas y el Editor de campos del componente `TFDQuery` a fin de ajustar la consulta, seleccionar los campos que deseamos tener y definir campos calculados. El procedimiento sería exactamente el mismo descrito entonces.*

[2]La finalidad de este componente es actuar como intermediario entre los componentes que representan conjuntos de datos, como `TFDQuery` y `TFDTable`, y los controles que componen la interfaz de usuario.

4.2.2 Interfaz de usuario

A continuación tendremos que diseñar la interfaz de usuario encargada de mostrar los datos procedentes del `TFDQuery`. Lo hacemos dando los pasos siguientes:

1. Agregamos un control `TPanel` al formulario, dando el valor `Top` a su propiedad `Align`. Eliminaremos el contenido de la propiedad `Caption` para quitar el texto que aparece en el panel.
2. Insertamos dentro del `TPanel` un `TCheckBox` que nos servirá para activar y desactivar la conexión. Usamos la propiedad `Caption` para establecer su título.
3. Por último introducimos en el formulario un `TDBGrid`, control análogo al `TGrid` de la FMX: una cuadrícula que puede ser conectada a un origen de datos. Usamos su propiedad `DataSource` para conectarlo al componente `TDataSource` que tenemos en el módulo de datos (véase la Figura 4.7).

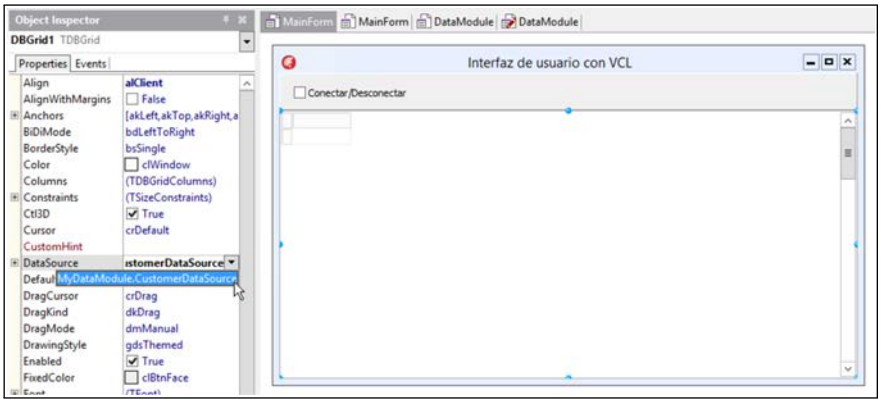


Figura 4.7 INTERFAZ DE USUARIO CON CUADRÍCULA Y `TCheckBox`

Solamente nos queda escribir el código que se encargará de habilitar o deshabilitar la conexión con la base de datos en respuesta al estado del `TCheckBox`. Haremos doble clic sobre este, para abrir el gestor correspondiente al evento `OnClick`, e introducimos el código mostrado a continuación:

```
1 procedure TForm3.CheckBox1Click(Sender: TObject);
2 begin
3   DBGrid1.DataSource.DataSet.Active := CheckBox1.Checked;
4 end;
```

Listado 4.1 Código asociado al control `TCheckBox`

Ahora ya podemos ejecutar el proyecto y comprobar su funcionamiento que, como cabría esperar, es casi idéntico al de la aplicación FMX que creábamos como ejercicio al final del capítulo previo (no hay posibilidad de filtrado por fecha y las columnas mostradas son otras). Las diferencias han sido puntuales: la inclusión de un `TDataSource` y el control que actúa como cuadrícula es un `TDBGrid` en lugar de un `TGrid`.

Aunque la conexión entre el origen de datos y la cuadrícula la hemos establecido manualmente, editando la propiedad `DataSource` del `TDBGrid`, también podríamos haber usado el **LIVEBINDINGS DESIGNER** para realizar dicha tarea tal y como hicimos en el capítulo previo.

4.3 Componentes VCL con conexión a datos

Al trabajar con la VCL y bases de datos, creando interfaces de usuario que facilitan la visualización y edición de información alojada en una base de datos, tradicionalmente se ha recurrido al uso de controles específicos con conexión a datos (*data-aware controls*).

Estos controles usan el prefijo `TDB` en sus nombres, por ejemplo `TDBEdit`, `TDBMemo`, `TDBNavigator` o `TDBGrid`. Todos ellos se alojan en una página específica de la Paleta de componentes: **DATA CONTROLS** (véase la Figura 4.8).

Los componentes con conexión a datos eran imprescindibles en la VCL, ya que los controles análogos sin el prefijo `TDB` no podían ser enlazados a un origen de datos por procedimientos RAD, debiendo recurrirse a escribir código de sincronización en ambos sentidos: desde el origen de datos al control y vice-

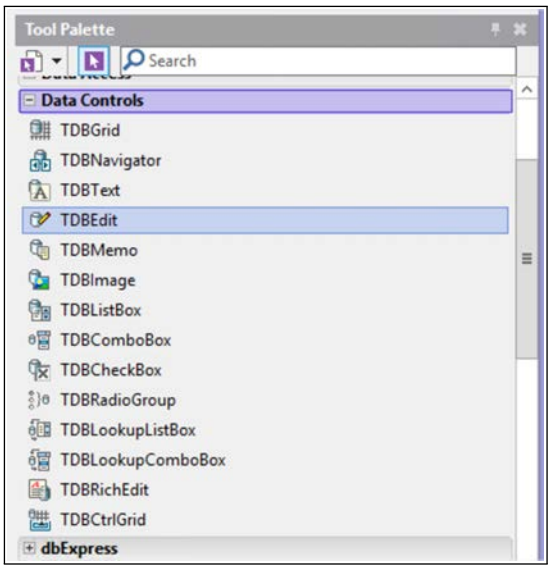


Figura 4.8 CONTROLES VCL CON CAPACIDAD PARA CONECTARSE A DATOS

versa. Esto, no obstante, ha cambiado completamente en las últimas versiones de Delphi, gracias a la introducción de los *LiveBindings*[3].

Todos los componentes que usan el prefijo `TDB` cuentan con una propiedad que no encontramos en el resto de controles: `DataSource`. La finalidad de esta es enlazar el control con un componente `TDataSource` que, a su vez, estaría vinculado al origen de datos, por ejemplo un `TDBQuery`. El `TDataSource` es un intermediario genérico, de forma que el origen de datos podría ser cualquier otro derivado de `TDataSet`. Dependiendo del tipo de control utilizado, además de la anterior también es usual encontrar la propiedad `DataField`. Su finalidad es enlazar el control con un campo concreto del origen de datos.

[3]Las primeras implementaciones de *LiveBindings* para la VCL, allá por la versión XE2, no siempre ofrecían la vía sencilla y rápida disponible para la FMX, pero esto es algo que se ha corregido en las versiones actuales

En principio, por tanto, y salvo que dependamos de controles vinculados a datos muy específicos, no hay muchas razones objetivas para decantarse por el uso de los controles tipo `TDB`, enlazados a un `TDataSource`, en lugar de usar los mismos controles que utilizaríamos en cualquier interfaz de usuario y recurrir al mecanismo de *LiveBindings*. De esta forma podemos seguir básicamente los mismos procedimientos de diseño para las interfaces de usuario, indistintamente de que la biblioteca de base sea VCL o FMX.

4.4 LiveBindings

Mostrar en una interfaz de usuario información procedente de un RDBMS o algún otro origen de datos, permitiendo también su edición, implica contar con algún tipo de conexión entre los controles de interfaz y los componentes de acceso a los datos. Los primeros estarían alojados en un formulario, mientras que los segundos normalmente residirán en un módulo de datos. Dicha conexión ha ser bidireccional, de forma que las acciones del usuario sobre los datos mostrados en la interfaz se transfieran adecuadamente al origen que los almacena.

El modelo descrito en la sección previa, disponible únicamente para componentes VCL, implica la duplicación de muchos de los controles con el simple objeto de agregar un vínculo a un `TDataSource`. Así tenemos el control `TDBEdit` que es funcionalmente equivalente al `TEdit`, pero dispone de las propiedades `DataSource` y `DataField`. Lo mismo se aplica a controles como `TDBMemo`, `TDBImage` o `TDBCheckBox`, equivalentes a `TMemo`, `TImage` y `TCheckBox`, respectivamente. De esta forma se limitan los elementos que es posible incluir en una interfaz de usuario con conexión a datos, únicamente pueden usarse de forma directa los controles de la página **DATA CONTROLS**, y por otra también se restringen las propiedades de esos controles que se vincularán a los datos.

Mediante *LiveBindings*, que es un motor de evaluación de expresiones en las que los operandos participantes son propiedades de objetos, el anterior esquema se expande a fin de permitir el uso de cualquier control y la conexión de cualquiera de sus propiedades a campos del origen de datos. Los objetos actúan como origen o bien como objetos controlados, pudiendo invertir dicho papel en caso necesario. Incluso se contempla la posibilidad de conexión entre propiedades de un mismo componente.

La principal ventaja de *LiveBindings*, ya apuntada antes, es que está disponible tanto en la VCL como en la FMX, por lo que es una opción de desarrollo homogénea con independencia de la biblioteca de componentes de interfaz que se elija.

4.4.1 Componentes LiveBindings esenciales

Desde la versión XE2 de Delphi, en la que se introdujo esta técnica para FMX y VCL, hasta hoy se han producido algunos cambios en los componentes asociados a *LiveBindings*. Los componentes `TBindScope` y `TBindScopeDB`, por ejemplo, fueron sustituidos por una jerarquía más flexible de componentes derivados de `TBaseLinkingBindSource`, como `TBindSourceDB`. Otros sin embargo se han mantenido, como es el caso de los componentes `TBindingsList` y `TBindNavigator`. La mayoría de ellos los encontramos en las páginas **LIVEBINDINGS** y **LIVEBINDINGS MISC** de la Paleta de componentes (véase la Figura 4.9.)

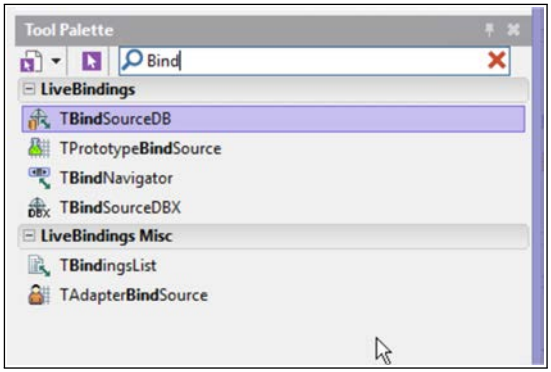


Figura 4.9 COMPONENTES FUNDAMENTALES DE LIVEBINDINGS

Por regla general no tendremos que agregar ni configurar manualmente estos componentes, dado que el **LIVEBINDINGS DESIGNER** nos permite establecer todas las conexiones con operaciones de arrastrar y soltar o bien mediante el asistente que conocimos en el capítulo previo. No obstante, analicemos qué hace el diseñador cuando establecemos un vínculo entre un origen de datos y controles de interfaz desde el citado diseñador.

Contexto del origen de datos

Para conectar un origen de datos, por ejemplo la información de un `TFDQuery`, con uno o más controles de interfaz lo primero que se precisa es un componente que facilite el contexto del origen de datos. Este lo aportará un componente derivado de la clase `TCustomBindSourceDB`. En la Paleta de componentes encontramos dos descendientes de dicha clase: `TBindDataSourceDB` y `TBindDataSourceDBX`. Usaríamos el primero para conectar con orígenes de datos FireDAC y el segundo si estuviésemos utilizando dbExpress. Nos centraremos en el primero ya que vamos a usar FireDAC.

El componente `TBindDataSourceDB` cuenta con una propiedad `DataSet` que le permite conectarse al `TFDQuery` antes citado o, alternativamente, a un `TFDTable` o algún otro derivado de `TFDDataSet`. A su vez, el componente `TBindDataSourceDB` se conectará a los controles de interfaz mediante la propiedad `DataSource` del enlace de *LiveBindings* apropiado.

En el **LIVEBINDINGS DESIGNER** un `TBindDataSourceDB` muestra, en su parte superior, el nombre del conjunto de datos a que está conectado, enumerando a continuación las columnas que obtiene del mismo (véase la Figura 4.10). Mediante arrastrar y soltar es posible conectar todo el conjunto de datos, con controles como `TGrid`, o bien columnas individuales, con controles como `TEdit`, `TCheckBox`, `TMemo`, etc.

Enlaces de LiveBindings

Los vínculos entre el componente `TBindSourceDB` y los controles que forman la interfaz se establecen mediante componentes de enlace *LiveBindings*, tales como `TLinkControlToField`, `TLinkListControlToField` o `TLinkGridToDataSource`. No encontraremos estos elementos en la Paleta de componentes. La herramienta encargada de gestionarlos es el diseñador asociado al componente `TBindingsList`.

> **NOTA**
>
> *Los componentes de la familia `TLinkXXXToYYY` actúan como intermediarios entre el `TBindSourceDB` y los controles de interfaz de usuario.*

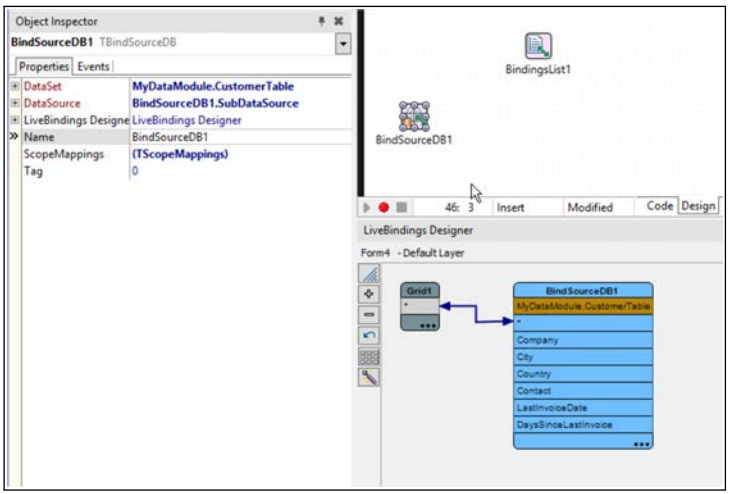


Figura 4.10 CONFIGURACIÓN DE UN `TBINDDATASOURCEDB`

Aunque podemos agregar al formulario un `TBindingsList` tomándolo de la Paleta de componentes, lo habitual es que se añada automáticamente en cuanto establezcamos un enlace visualmente, desde el **LIVEBINDINGS DESIGNER**. Al hacer doble clic sobre ese componente se abrirá la ventana mostrada en la Figura 4.11. El panel de la izquierda permite agrupar los enlaces por categorías, mientras que el de la derecha muestra el nombre y descripción de cada uno de los enlaces existentes en la categoría elegida.

En lugar de crear el enlace entre el `TBindSourceDB` y los controles de interfaz mediante operaciones de arrastrar y soltar, que generalmente es lo más cómodo, podemos usar el primer botón de la barra de herramientas de esta ventana para crearlos manualmente. El primer paso sería seleccionar el tipo de enlace que necesitamos, eligiéndolo de la lista a la que da paso el citado botón (véase la Figura 4.12). El segundo la configuración del enlace usando el Inspector de objetos.

Las propiedades que generalmente nos interesará configurar de un enlace son las indicadas a continuación:

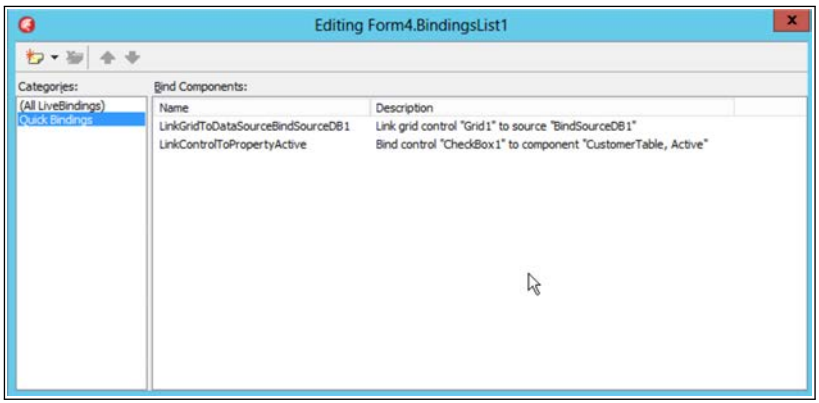


Figura 4.11 DISEÑADOR ASOCIADO AL COMPONENTE TBINDINGSLIST

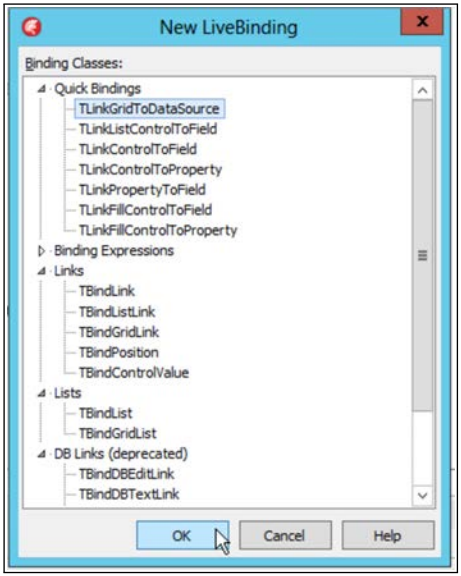


Figura 4.12 LISTA DE CLASES DE ENLACES *LiveBindings*

- `DataSource`: Apuntará al `TBindSourceDB` que establece el contexto del origen de datos que se quiere utilizar.
- `Control`: Establecerá el control que se enlazará al contexto de origen de datos indicado por la propiedad anterior. Puede ser cualquiera de los controles existentes en el formulario.
- `FieldName`: En caso de que el control no sea de tipo cuadrícula, sino que se vincula a una sola columna del origen de datos, esta propiedad permite elegir dicha columna.
- `Direction`: Mediante esta propiedad se determina cuál será el sentido en el que fluirán los datos a través del enlace. Las posibles configuraciones son tres: del origen de datos hacia el control (`linkDataToControl`), del control hacia el origen de datos (`linkControlToData`) o bidireccional (`linkBidirectional`). Esta última configuración es la más habitual y por ello es la que se utiliza por defecto.

4.4.2 Adición y enlace de controles

Ya conocemos tres métodos distintos para agregar controles a un formulario y enlazarlos con un origen de datos:

- Usando el asistente del panel **LIVEBINDINGS DESIGNER**, concretamente las opciones **LINK A CONTROL WITH A FIELD** y **LINK A GRID WITH A DATA SOURCE**. Estas añaden automáticamente el control al formulario si es preciso, insertando también el `TBindingsList` si no existiese ya. A continuación se agrega el componente de enlace adecuado y se configura.
- Agregando manualmente el control a la interfaz de usuario y a continuación, también usando el **LIVEBINDINGS DESIGNER**, establecer el vínculo con operaciones de arrastrar y soltar. Estas añaden al `TBindingsList` el componente `TLinkXXXToYYY` apropiado y lo configuran, estableciendo el vínculo entre el origen de datos y el control.
- Usando el diseñador asociado al `TBindingsList` para crear manualmente el componente de enlace y configurarlo mediante el Inspector de objetos.

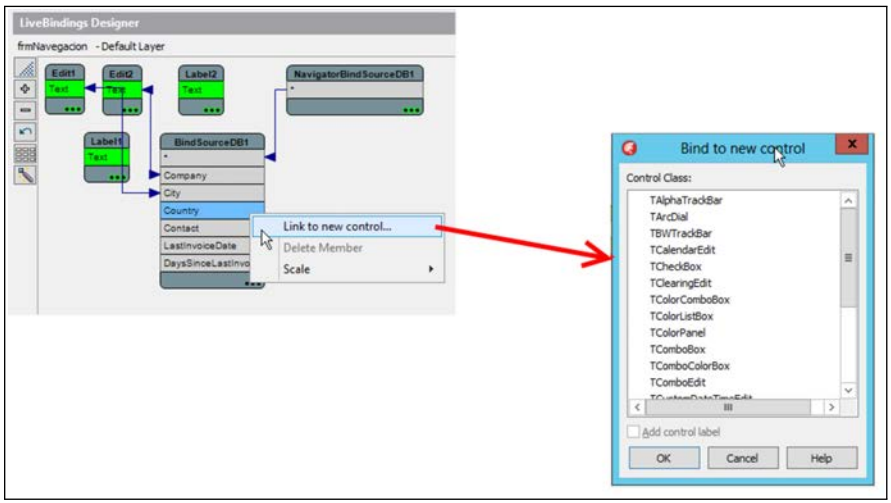


Figura 4.13 INSERTAR Y ENLAZAR UN NUEVO CONTROL A UN CAMPO DE DATOS

Existe al menos una vía adicional para realizar esta misma tarea. Una vez que disponemos del `TBindSourceDB` en el **LIVEBINDINGS DESIGNER**, podemos usar el menú contextual de cualquier columna para ejecutar la opción **LINK TO NEW CONTROL**. Esta abrirá una lista de controles disponibles para el tipo de campo elegido (véase la Figura 4.13). No tenemos más que elegir el que nos interese para agregarlo automáticamente la formulario y enlazarlo con la columna elegida.

4.5 Diseño de una interfaz con controles simples y navegación

En los ejercicios propuestos hasta ahora hemos recurrido a la visualización de datos en una cuadrícula, representada por el control `TGrid` o `TDBGrid` según los casos. Las cuadrículas permiten mostrar en una estructura tabular un conjunto

de filas y columnas, facilitando asimismo las operaciones de edición de datos. No obstante, no siempre es la opción más apropiada para el diseño de una interfaz de usuario conectada a datos.

Si la información a mostrar no se ajusta a una estructura tabular, por ejemplo al existir campos de texto extensos o imágenes, la mejor alternativa suele ser el diseño de un formulario con controles simples[4], que permiten operar sobre una única fila de datos, incluyendo los controles de navegación apropiados. Veamos cómo hacerlo a través de un nuevo ejercicio.

EJEMPLO 4.2 `VCLLiveBindings.dproj`

Puedes encontrar este proyecto completo en la carpeta `VCLLiveBindings`. Puedes abrirlo y ejecutarlo directamente ya que no existe conexión con origen de datos externo alguno.

4.5.1 Origen de los datos

A fin de concentrarnos en el diseño de la interfaz y la conexión de los controles a los datos, que es lo que nos interesa en este caso, vamos a prescindir del módulo de datos y de los componentes FireDAC para acceso a una base de datos concreta. En su lugar usaremos el componente `TPrototypeBindSource`, cuya finalidad es generar datos de prueba para escenarios como el actual.

Tras agregar ese componente al formulario abriremos el Editor de campos, con la opción **FIELDS EDITOR** de su menú contextual, y a continuación haremos clic en el primer botón del editor vacío, haciendo así aparecer la lista de columnas disponibles. Como se aprecia en la Figura 4.14, tenemos columnas de cualquier tipo: cadenas de caracteres, números enteros y en punto flotante, fechas, booleanos, mapas de bits, etc.

Seleccionaremos las columnas siguientes para añadirlas al Editor de campos: `ContactBitmaps`, `ContactNames`, `ContactTitles`, `Currency`

[4]Entendiendo en este contexto que un control *simple* es aquél que se vincula a una sola columna del origen de datos.

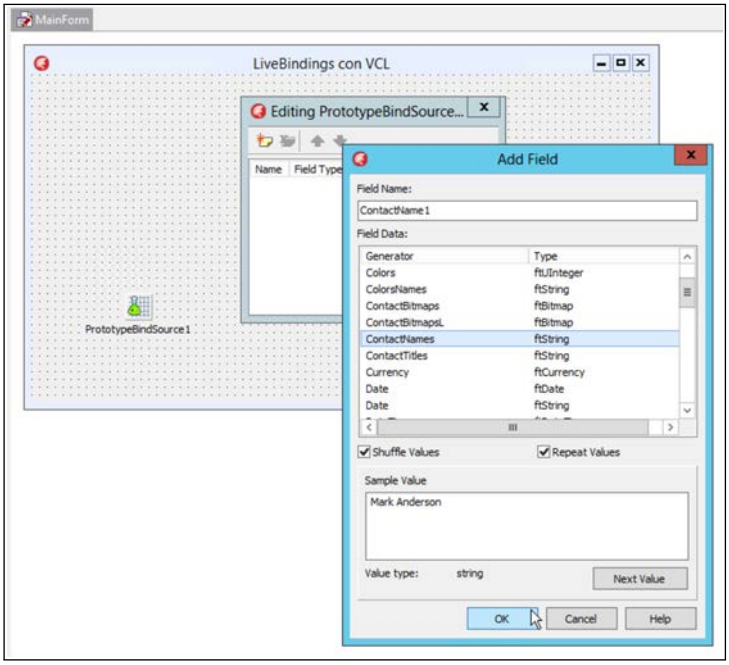


Figura 4.14 PODEMOS SELECCIONAR LAS COLUMNAS A INCLUIR

y `LoremIpsum` (tipo `ftTstrings`). Con esto tenemos a nuestra disposición un grupo de campos con distintos tipos: textos, números, imágenes y conjunto de líneas de texto. Si abrimos el **LIVEBINDINGS DESIGNER** los encontraremos como columnas en el componente `TPrototypeBindSource`.

4.5.2 Diseño de la interfaz de usuario

Mediante el menú contextual asociado a cada columna usaremos la opción **LINK TO NEW CONTROL** para agregar los controles al formulario y, al mismo tiempo, configurar el enlace con el origen de datos.

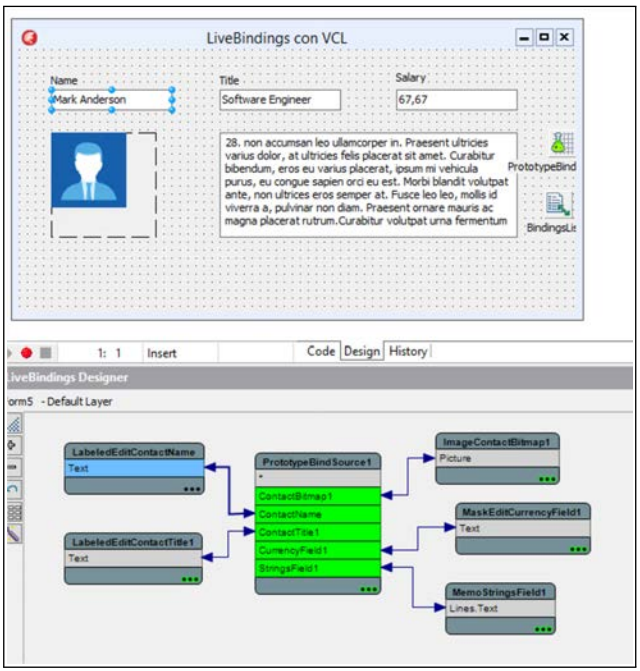


Figura 4.15 DISPOSICIÓN DE LOS CONTROLES Y ENLACES AL ORIGEN DE DATOS

Elegiremos un `TImage`, dos `TLabeledEdit`, un `TMaskEdit` y un `TMemo`, respectivamente al orden en que añadimos las columnas. La disposición de los controles podría ser la que se muestra en la Figura 4.15.

A continuación podríamos abrir el editor del componente `TBindingsList` y examinar los objetos que se han añadido para establecer los enlaces. Todos ellos serán de tipo `TLinkControlToField`. Lo que cambiará de uno a otro será los valores asignados a las propiedades `Control` y `FieldName`.

4.5.3 Controles de navegación

Los controles que hemos introducido en el formulario servirán para mostrar las columnas de una fila de datos, por lo que será preciso añadir algún control que

facilite la navegación por las filas que contuviese el conjunto de datos. Para ello recurriremos al menú contextual del componente `TPrototypeBindSource` y elegiremos la opción **ADD NAVIGATOR**. Esta insertará en el formulario un control `TBindNavigator` ya vinculado al origen de datos. La única propiedad que cambiaremos de dicho control será `Align`, a la que daremos el valor `alBottom` para colocarlo en la parte inferior del formulario.

Con esto ya hemos terminado el desarrollo de este ejercicio. Podemos ejecutar el proyecto y el resultado debería ser parecido al que se muestra en la Figura 4.16. Usando los controles de navegación podemos movernos por las filas del conjunto de datos, así como eliminar, agregar y modificar filas.

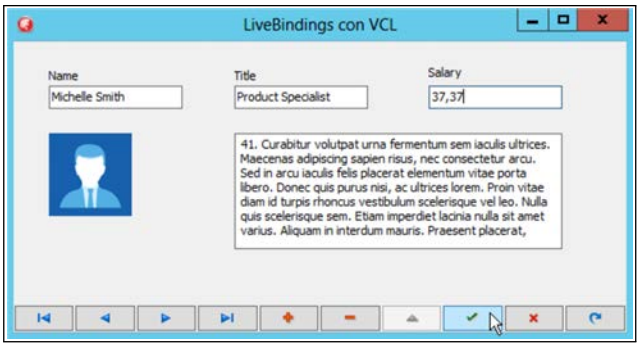


Figura 4.16 LA APLICACIÓN VCL EN FUNCIONAMIENTO

Aunque en este caso hemos utilizado un origen de datos artificial, representado por el componente `TPrototypeBindSource`, el procedimiento habría sido idéntico si hubiésemos partido de un proyecto con un módulo de datos y componentes FireDAC conectados a una base de datos real, como hicimos en ejercicios previos.

4.6 Otras configuraciones de datos

Hasta ahora hemos aprendido a construir, tanto con la VCL como la FMX, formularios basados en cuadrículas y también en controles simples, en el caso de

la VCL recurriendo a los antiguos `TDBxxx` y también, en el apartado previo, usando *LiveBindings*. Básicamente estas son todas las combinaciones posibles que podemos encontrar. A partir de ellas, no obstante, se construyen configuraciones más específicas. Estas son el objeto de la última sección del presente capítulo.

> **NOTA**
>
> *A partir de este punto, tanto en el capítulo actual como en los posteriores, siempre basaremos los ejercicios en proyectos de tipo* **MULTI-DEVICE APPLICATION** *con FMX y usando LiveBindings. Los procedimientos para trabajar con la VCL, en cuanto a la conexión a datos y composición de la interfaz, serían equivalentes según se ha descrito en las secciones previas.*

La naturaleza del modelo de datos relacional, con sus reglas de normalización, contribuye a que en las bases de datos existan referencias entre las tablas. Dichas relaciones pueden ser de tipo uno a uno, de forma que a una fila de una tabla le corresponde solo una fila de otra, o bien uno a muchos, en cuyo caso tenemos una relación de tipo maestro/detalle. Veamos cómo gestionar este tipo de relaciones al diseñar una interfaz de usuario con acceso a datos en Delphi.

4.6.1 Relaciones maestro/detalle

En este tipo de relación a una fila de una tabla le corresponden una o más filas de otra. La primera tabla es la maestra y la segunda es la tabla de detalle. Los casos más típicos son los de las tablas de facturas y líneas de factura o de pedidos y líneas de pedido. La tabla de facturas almacena toda la información de cabecera de una factura: nombre, dirección e identificación del cliente, fecha en que se emite, importe total antes y después de impuestos, etc. Todos esos datos no hay que repetirlos para cada línea de detalle de la factura, en la que se especifica el artículo o servicio que está facturándose, el precio unitario, la cantidad, etc. La separación en dos tablas de una entidad física única, como es la factura, permite reducir la información redundante. Lógicamente, ha de existir una columna común que relacione a ambas tablas, permitiendo localizar todas las líneas de una factura.

Al trabajar con FireDAC este tipo de relaciones se definen atendiendo a los siguientes pasos:

- Cada tabla, la maestra y la de detalle, estaría representada por su propio componente `TFDQuery`, conectados ambos al mismo `TFDConnection`.
- El `TFDQuery` asociado a la tabla de detalle incluirá en la consulta un parámetro de filtrado de las filas, de forma que solo se obtengan aquellas que correspondan a la fila actualmente activa en la tabla maestra.
- El `TFDQuery` de la tabla de detalle se vinculará con la tabla maestra mediante la propiedad `MasterSource`. Para ello se utilizará un componente `TDataSource` como intermediario entre ambas tablas.
- En caso necesario se usarán las propiedades `MasterFields` y `DetailFields` del `TFDQuery` de detalle para establecer el campo común que vincula a ambas tablas, indicando los campos de la tabla maestra y de detalle que lo forman. En la mayoría de los casos esto no es preciso ya que dicho campo tendrá el mismo nombre que el parámetro usado en la consulta del incluida en el `TFDQuery`.

Una vez que se ha establecido la relación apropiada entre los componentes de datos, el diseño de la interfaz se llevará a cabo según los procedimientos que ya conocemos. Lo habitual es que la información común, la de la tabla maestra, se disponga en un formulario usando controles individuales, mientras que los datos de detalle se recogen en algún tipo de cuadrícula.

4.6.2 Campos de búsqueda

Cuando la relación entre dos tablas es de uno a uno, no de uno a muchos, lo que suele necesitarse es que un campo de una tabla sirva para buscar un dato en la otra. Es una técnica común conocida como *lookup field* y FireDAC nos ofrece las opciones necesarias para definirlos.

En la sección 3.7.4, durante el desarrollo de un ejercicio, se describió cómo definir un campo calculado. Entre las opciones disponibles en la ventana usada para ello encontraremos la que permite crear campos de búsqueda.

4.6.3 En la práctica

Veamos cómo componer una interfaz de usuario en la que exista tanto una relación maestro/detalle como un campo de búsqueda. Volveremos a apoyarnos en la base de datos `DBDEMOS`, ya que es una base de datos Access local que no precisa de mayor configuración ni infraestructura para su uso.

> **EJEMPLO 4.3** `MaestroDetalle.dproj`
>
> Puedes encontrar este proyecto completo en la carpeta `MaestroDetalle`. Su funcionamiento depende de la base de datos Access `DBDEMOS`, por lo que necesitarás realizar los ajustes necesarios en el archivo de definiciones de FireDAC.

Una vez iniciado el nuevo proyecto, de tipo **MULTI-DEVICE APPLICATION**, los pasos que debemos seguir son los descritos a continuación, en los apartados siguientes.

Preparación del módulo de datos

Agregaremos el proyecto un nuevo módulo de datos y, sirviéndonos del Explorador de datos, arrastraremos hasta él las tablas `Orders`, `Items` y `Customer` de la base de datos `DBDEMOS`. Esto agregará un `TFDConnection`, con los parámetros de conexión a la base de datos, y tres `TFDQuery` representando a las tres tablas.

Tenemos que añadir un componente `TDataSource`, así como el habitual `TFDGUIxWaitCursor`. Usaremos la propiedad `DataSet` del primero para enlazarlo con la tabla `Orders`. Esto nos permitirá definir la relación maestro/detalle entre los componentes `OrdersTable` e `ItemsTable`. Para ello haremos lo siguiente:

1. Abrimos el menú contextual del `TFDQuery` asociado a la tabla `Items` y elegimos la opción **QUERY EDITOR** para abrir el editor de consultas.

2. Agregamos a la consulta ya existente en el editor un filtro de búsqueda, mediante la cláusula `WHERE OrderNo = :OrderNo`, tal y como se aprecia en la parte derecha de la Figura 4.17.

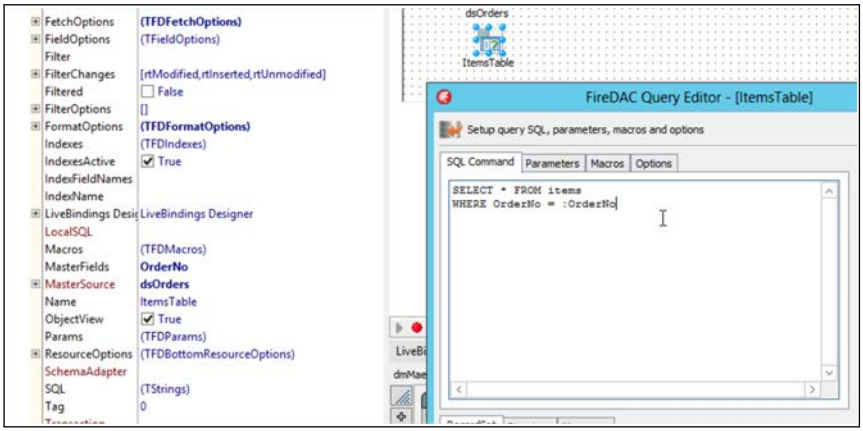


Figura 4.17 ESTABLECEMOS EL VÍNCULO ENTRE TABLA MAESTRA Y DE DETALLE

3. Cerramos el Editor de consultas y recurrimos al Inspector de objetos para establecer la propiedad `MasterSource` del mismo componente, seleccionando el `TDataSource` que anteriormente habíamos vinculado a la tabla de pedidos.

4. Opcionalmente podemos fijar el valor de las propiedades `MasterFields` y `DetailFields`. Estas propiedades tienen asociado un editor como el de la Figura 4.18, en el que podemos seleccionar columnas de las lista de la izquierda y agregarlas a la de la derecha. En este caso solamente se precisaría la columna `OrderNo` que es el dato común entre la tabla maestra y la de detalle: el número de pedido.

A continuación tenemos que definir el campo de búsqueda que nos permitirá obtener, a partir del código de cliente que se almacena en la tabla `Orders`, el nombre de la empresa cliente alojada en la tabla `Customer`. En este caso el procedimiento se compone de los siguientes pasos:

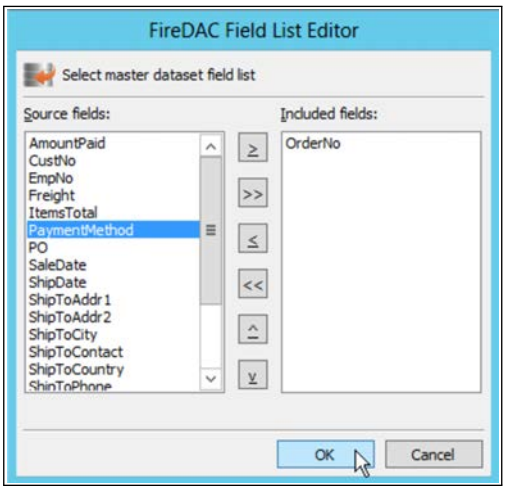


Figura 4.18 SELECCIÓN DE LOS CAMPOS PARA EL VÍNCULO MAESTRO/DETALLE

1. Abrimos el menú contextual del componente `OrdersTable` y seleccionamos la opción **FIELDS EDITOR** a fin de abrir el Editor de campos. Este estará inicialmente vacío.

2. Mediante la opción **ADD FIELD** del menú contextual del Editor de campos agregamos los campos que vamos a necesitar: `OrderNo`, `SaleDate` y `CustNo`. Estos contienen el código de pedido, fecha de la venta y código del cliente, respectivamente.

3. Usando la opción **NEW FIELD** del Editor de campos accedemos al cuadro de diálogo del mismo nombre, en el que procederemos a configurar el campo de búsqueda.

4. Introducimos `CustomerName` como nombre del campo y seleccionamos de la lista desplegable **TYPE** el tipo `String`.

5. En el apartado **FIELD TYPE**, que contiene las opciones de tipo de campo, seleccionaremos el tipo **LOOKUP**. Esto activará los controles del apartado **LOOKUP DEFINITION**, situado en la parte inferior de la ventana.

6. La configuración del campo de búsqueda será la mostrada en la Figura 4.19. Con ella establecemos que usando el campo `CustNo` de la tabla actual (`OrdersTable`) se buscará coincidencia con el campo `CustNo` en el componente `CustomerTable`, recuperándose el contenido de su columna `Company`.

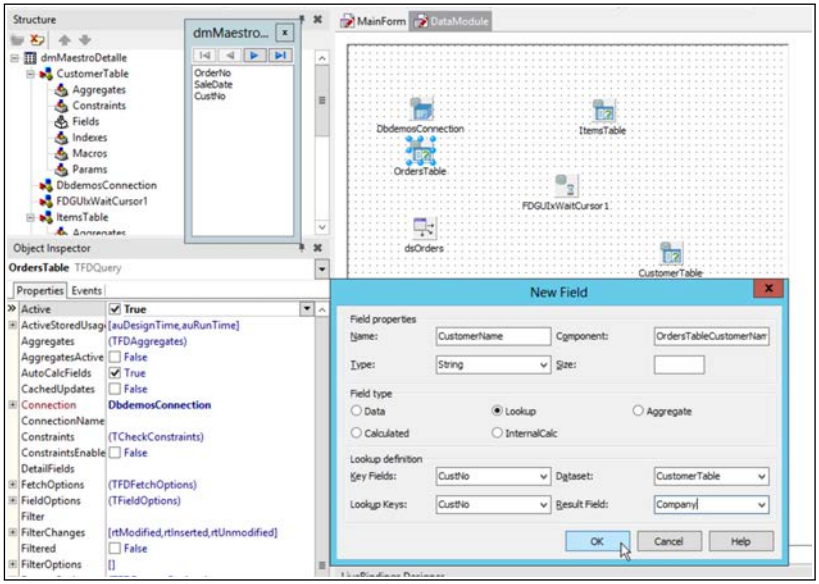


Figura 4.19 CONFIGURACIÓN DEL CAMPO DE BÚSQUEDA

Lo que obtendríamos en el campo CustomerName, por tanto, sería el nombre de la empresa a la que corresponde el pedido. Con esto tenemos todo lo necesario en cuanto a configuración de acceso a datos. Solamente quedaría dar el valor `True` a la propiedad `Active` de los `TFDQuery`, a fin de que podamos ver durante el diseño la información sobre la que estamos trabajando.

Diseño de la interfaz de usuario

En el proyecto tenemos un formulario vacío, al que vamos a añadir los controles necesarios para recorrer los pedidos y las líneas que le correspondan. Lo primero

que debemos hacer es usar la opción **FILE—USE UNIT** para agregar una referencia al módulo de datos, de forma que podamos acceder a los componentes alojados en él.

Utilizando el **LIVEBINDINGS DESIGNER** enlazamos las columnas de la tabla de pedidos con dos controles `TEdit` y un `TCalendarEdit`. También agregaremos un `TBindNavigator` asociado a esa misma tabla. Recuerda que no tienes más que abrir el menú contextual del `TBindSource` y elegir la opción **ADD NAVIGATOR**. También introduciremos un control `TGrid`, vinculándola con la tabla de líneas de pedido (`ItemsTable`). El enlace entre columnas de datos y los controles introducidos en la interfaz de usuario son los mostrados en la Figura 4.20.

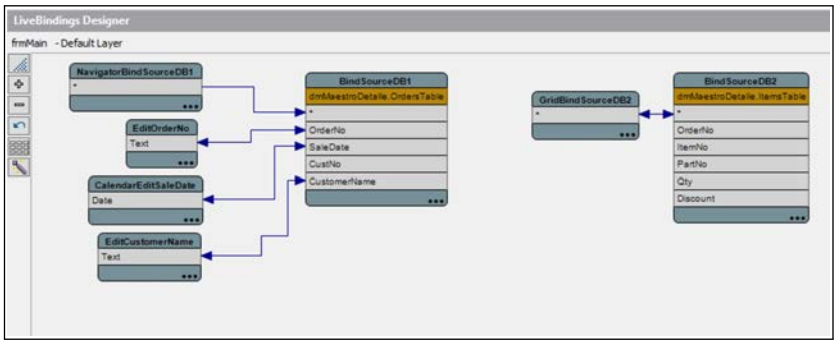


Figura 4.20 ENLACES ENTRE COLUMNAS DE DATOS Y CONTROLES DE INTERFAZ

Puesto que toda la lógica que relaciona unas tablas con otras ya ha sido establecida en el módulo de datos, en el formulario no es preciso hacer nada más, una vez agregados los controles, para poder navegar por los pedidos y ver por cada uno de ellos la fecha, cliente a que corresponde y los productos incluidos en el mismo. En la Figura 4.21 se muestra la aplicación en ejecución.

> **NOTA**
>
> *Dado que no hemos agregado código alguno al proyecto, el formulario solamente mostrará los datos si hemos dado el valor `True` a la propiedad `Active` de los `TFDQuery` durante la fase de diseño.*

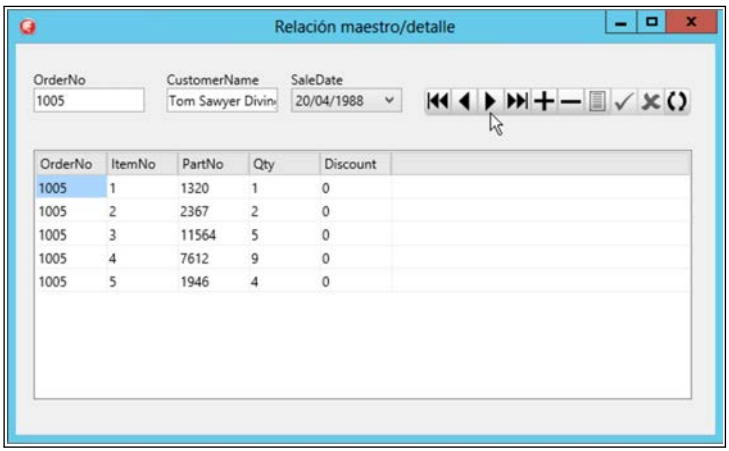


Figura 4.21 EL PROGRAMA CON LA INTERFAZ MAESTRO/DETALLE EN FUNCIONAMIENTO

4.7 Resumen

En este capítulo hemos conocido las diferencias fundamentales entre la VCL y la FMX a la hora de desarrollar un proyecto Delphi con conexión a bases de datos, así como los procedimientos a seguir en cada uno de los casos. La conclusión final sería que el uso de *LiveBindings*, con independencia de la biblioteca de componentes elegida, nos ofrece la vía más flexible para establecer los enlaces entre los datos y la interfaz, siendo además una vía común a VCL y FMX, por lo que sería más sencillo cambiar de una a otra.

NOTA

Es posible utilizar LiveBindings para crear otros tipos de enlaces entre propiedades de componentes, sin que estén implicadas bases de datos, usando también expresiones a medida. Puedes encontrar más información

sobre el uso en estos contextos en el libro La guía de Delphi, del mismo autor y editorial.

Además hemos aprendido a diseñar varios de los tipos de interfaces con conexión a datos más habituales: usando cuadrículas de datos, empleando controles simples y combinando ambos enfoques para representar relaciones maestro/detalle e incluir campos de búsqueda.

Con esta base, en los capítulos siguientes abordaremos casos más específicos de aplicaciones Delphi con acceso a bases de datos locales, incluyendo el uso de InterBase embebido, bases de datos de escritorio y bases de datos en memoria, entre otros.

Capítulo 5

APLICACIONES CON INTERBASE EMBEBIDO

A diferencia de SQLite o Microsoft Access, que son los tipos de bases de datos que hemos utilizado en los ejercicios de capítulos previos, InterBase es un verdadero RDBMS. Esto implica que la base de datos puede contener objetos adicionales, como las vistas, las funciones, los procedimientos almacenados y los disparadores, que no son habituales en bases de datos de escritorio.

InterBase puede ser instalado en un servidor y actuar como cualquier otro RDBMS, como puede ser SQL Server u Oracle, pero también puede utilizarse en otras configuraciones menos habituales para un servidor de datos. Una de ellas es el uso embebido en las aplicaciones, de forma que el propio motor que ges-

tiona los datos es distribuido con el programa, no siendo precisas instalaciones adicionales ni acceso remoto a los datos (como es habitual en un servidor).

En este capítulo vamos a conocer el procedimiento que seguiríamos para utilizar InterBase embebido en un proyecto desarrollado con Delphi, poniéndolo en práctica en un ejercicio cuyo resultado podremos ejecutar tanto en un ordenador de escritorio como en dispositivos móviles.

5.1 InterBase ToGo

InterBase, como servidor de datos, puede ser instalado en Windows, GNU/Linux y OS X, estando disponible tanto en versiones de 32 como de 64 bits. Como ocurre al trabajar con cualquier otro RDBMS alojado en un servidor, un equipo distinto del que ejecuta la aplicación, nuestro programa utilizaría un controlador FireDAC para comunicarse con el software cliente del RDBMS que, a su vez, se comunicaría a través de red con el servidor para ejecutar las tareas encomendadas.

También podemos utilizar InterBase como servidor datos local, configuración en la que el software cliente y el servidor residen en el mismo equipo. La comunicación entre ambos componentes, por tanto, puede ser directa en lugar de a través de TCP/IP. Este modo de trabajo local, para el cual precisaríamos la licencia *Developer* del producto, nos permitirá crear nuevas bases de datos, definir las estructuras de tablas, vistas, etc., e introducir la información que se precise. Todas estas son tareas que, por regla general, efectuaremos en un ordenador de escritorio con un servidor local de InterBase.

Finalmente tenemos InterBase ToGo que, en esencia, es el núcleo del motor que gestiona las bases de datos InterBase, un módulo redistribuible que hace posible que las aplicaciones desarrolladas con Delphi accedan a bases de datos InterBase directamente, sin software de comunicaciones cliente ni un servidor de datos. InterBase ToGo está disponible para múltiples sistemas operativos, incluyendo los OS móviles iOS y Android, lo que le convierten en la edición más multiplataforma de InterBase.

InterBase ToGo, por tanto, es ideal para el uso embebido (o empotrado) en aplicaciones móviles, aportando muchos de los beneficios de trabajar con un verdadero RDBMS pero sin los costes y complejidades de su instalación y ad-

ministración. Únicamente hay que incluir, al desplegar la aplicación, el motor de InterBase ToGo, el archivo que contiene la base de datos y el archivo de licencia correspondiente.

5.1.1 Licencias ToGo e IBLite

InterBase ToGo es, como componente software, un producto único. No obstante puede ser utilizado bajo dos tipos diferentes de licenciamiento, a los que se denomina comercialmente ToGo e IBLite. La primera tiene un cierto coste económico, mientras que la segunda es gratuita. Las diferencias fundamentales entre ambas ediciones se indicaron en la sección 2.2.1 (capítulo 2).

Delphi incluye una versión de desarrollo de InterBase ToGo que nos permite utilizar este producto durante un tiempo limitado, con el objetivo de facilitar la realización de pruebas. Para distribuir aplicaciones con este RDBMS tendríamos que adquirir la licencia correspondiente o, en su defecto, optar por IBLite siempre que nuestro proyecto pueda adecuar su funcionamiento a las limitaciones de esta versión gratuita.

> **NOTA**
>
> *En `http://docwiki.embarcadero.com/RADStudio/XE8/en/IBLite_and_IBToGo_Licensing_in_RAD_Studio` se detalla el proceso de registro a seguir para obtener una licencia gratuita de IBLite, así como para utilizar la licencia de desarrollo de la edición ToGo.*

5.2 Herramientas InterBase para desarrollo

Crear un proyecto en el que van a utilizarse bases de datos InterBase requerirá que durante la fase de desarrollo, en el equipo donde estemos trabajando, podamos operar sobre dichas bases de datos. Para ello lo habitual es que utilicemos el *Local InterBase Server* que se incluye con Delphi. Este es un servidor

local mediante el que podemos conectar con cualquier base de datos InterBase y trabajar con ella mediante las herramientas para desarrollo.

5.2.1 InterBase Manager

La primera herramienta con la que tendremos que familiarizarnos será el *InterBase Manager*, una utilidad mediante la que se controla el estado del servidor local de InterBase y se configuran sus propiedades. La interfaz de esta consola de administración es la mostrada en la Figura 5.1.

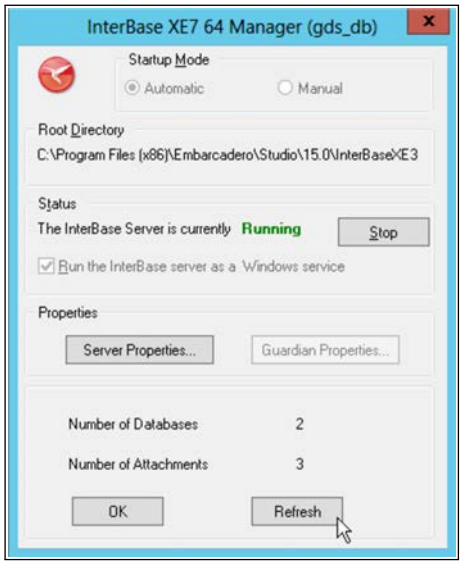


Figura 5.1 INTERFAZ DEL INTERBASE MANAGER

Lo habitual es que InterBase se inicie automáticamente cada vez que se ponga en marcha el sistema, pero mediante la opción **MANUAL** del apartado **STARTUP MODE** podemos optar por un inicio manual[1]. En este caso utilizaríamos el botón

[1]Para poder cambiar el modo de inicio es necesario que el servidor esté parado, por lo que primero haríamos clic en el botón **STOP** y después modificaríamos la opción de inicio.

adjunto al apartado **STATUS** para iniciar o detener el servidor local de datos, a demanda.

El botón **SERVER PROPERTIES** simplemente abre un panel en la parte inferior de la ventana, indicando el número de bases de datos y conexiones activas. Mediante el botón **GUARDIAN PROPERTIES** podremos saber el número de reinicios que ha sufrido el servidor de datos. Este botón será inaccesible en caso de que el servidor local de InterBase esté ejecutándose como un servicio. El guardián es un proceso que vigila constantemente el estado del servidor y, en caso de que haya problemas, procede a reiniciarlo.

Fundamentalmente usaremos esta herramienta para asegurarnos de que el servidor está en funcionamiento y, en caso necesario, proceder a iniciarlo. De esta forma podremos acceder a él desde el entorno de Delphi, nuestras aplicaciones o la consola de InterBase.

5.2.2 La consola de InterBase

Una vez que el servidor local de InterBase está en funcionamiento, la herramienta a la que recurriremos habitualmente es la consola de InterBase. La encontrarás buscando `ibconsole` en el menú/pantalla **INICIO** de Windows. Desde esta utilidad podremos comunicarnos tanto con el servidor local como con cualquier servidor InterBase remoto. No tenemos más que registrar el servidor con el que deseemos trabajar, usando para ello las opciones del menú contextual asociado al nodo **INTERBASE SERVER** (raíz del panel izquierdo).

Tras conectar con una base de datos, por ejemplo la clásica `Employee`[2] que se incluye como ejemplo con InterBase, podremos examinar todos los objetos que contiene, acceder a sus propiedades y, si fuese preciso, modificarlos. También encontraremos en el menú contextual asociado a cada tipo de objeto las opciones necesarias (véase la Figura 5.2) para crearlos, por ejemplo definiendo nuevas tablas, vistas, procedimientos almacenados, etc.

Asimismo desde la consola también es posible efectuar tareas de mantenimiento sobre las bases de datos, efectuando copias de seguridad, gestionando los privilegios de los usuarios o analizando los datos de rendimiento asociados

[2]Las credenciales por defecto para acceder a esta base de datos son `SYSDBA` y `MASTERKEY`.

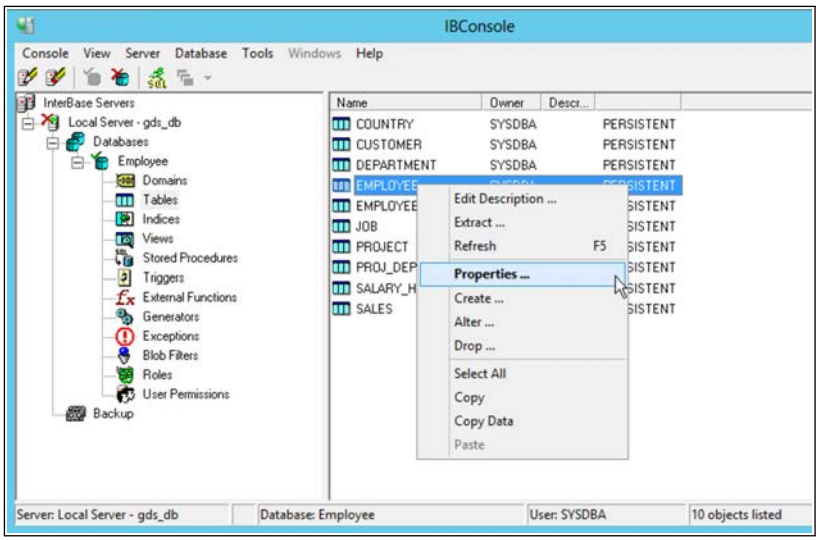


Figura 5.2 CONSOLA DE INTERBASE

a la misma. Todas estas opciones nos permitirán preparar la base de datos que vayamos a utilizar desde nuestra aplicación, comenzando por la creación de la propia base de datos, la definición de las tablas y demás objetos, creación de usuarios y, finalmente, obtención del archivo que contiene toda la información. Este será un archivo con extensión `.gdb`, totalmente compatible con InterBase ToGo e IBLite.

> **NOTA**
>
> *La información relativa a las opciones ofrecidas por la consola de InterBase puedes encontrarla en* `http://docs.embarcadero.com/products/interbase/IBXE3Update3/ToGoQuickStart/Introduction.htm`

En las secciones posteriores de este capítulo, en las que aprenderemos a acceder a bases de datos InterBase con FireDAC, utilizaremos la citada base de

datos de ejemplo `Employee`. No obstante, los procedimientos a seguir serían idénticos con cualquier base de datos que hubiésemos creado a medida para nuestro proyecto.

5.3 InterBase y FireDAC

FireDAC incluye los controladores necesarios para comunicarse con InterBase, indistintamente de cuál sea su versión: ToGo, IBLite, servidor local o remoto. No tenemos más que agregar el componente `TFDPhysIBDriverLink` a nuestro proyecto y establecer los parámetros adecuados para que el componente `TFDConnection` use dicho controlador. También podemos arrastrar y soltar desde el Explorador de datos, como hemos hecho en capítulos previos, dejando que el diseñador incluya todos los elementos necesarios.

Además de los componentes de uso común que ya conocemos, como el citado `TFDConnection` o el componente `TFDQuery`, en la página **FIREDAC SERVICES** de la Paleta de componentes encontraremos también varios específicos para trabajar con InterBase. La finalidad de estos, brevemente, es la indicada a continuación:

- `TFDIBConfig`: Permite poner una base de datos *offline*, para efectuar tareas de mantenimiento, y *online*, lista para trabajar de nuevo, así como modificar algunos parámetros de configuración como el tamaño de la caché, el dialecto de SQL, etc.
- `TFDIBValidate`: Facilita el acceso a las operaciones de validación y reparación de bases de datos.
- `TFDIBBackup` y `TFDIBRestore`: Mediante estos dos componentes es posible realizar una copia de seguridad de una base de datos y recuperarla si fuese necesario.
- `TFDIBSecurity`: Los métodos de este componente hacen posible agregar, modificar y eliminar cuentas de usuario de la base de datos, entre otras operaciones relacionadas con la seguridad.

Todas estas operaciones pueden ser completadas desde la consola de InterBase, pero una vez desplegada la información es habitual que dicha herramienta

no esté disponible. Con los componentes anteriores es posible integrar la misma funcionalidad en una aplicación. En la mayoría de los casos no hay más que facilitar los parámetros de acceso a la base de datos e invocar al método adecuado.

NOTA

Los componentes `TFDIBxxx` permiten operar tanto sobre bases de datos InterBase como Firebird.

5.4 Caso práctico

Veamos a través del desarrollo de un ejercicio práctico cómo utilizar InterBase embebido en una aplicación. Concretamente utilizaremos IBLite para facilitar la visualización de información almacenada en la base de datos de ejemplo `Employee`.

Dado que el objetivo es comprobar cómo podemos incluir IBLite en el despliegue del proyecto a distintos dispositivos, incluyendo móviles, iniciaremos un nuevo proyecto de tipo **MULTI-DEVICE APPLICATION**, usando la plantilla que genera un proyecto con un formulario vacío. Desde este punto de partida seguiremos los procedimientos descritos en los siguientes apartados para construir la aplicación.

EJEMPLO 5.1 `IBEmbebido.dproj`

Puedes encontrar este proyecto completo en la carpeta `IBEmbebido`. Es posible que tengas que realizar ajustes en la configuración de despliegue de la aplicación, según el sistema en que quieras ejecutarlo y su configuración.

5.4.1 Configuración de la conexión

Añadiremos el proyecto un módulo de datos, a fin de separar los componentes FireDAC de la interfaz de usuario. El primer componente que insertaremos en el módulo de datos será un `TFDConnection`, haciendo doble clic sobre él para abrir la ventana de configuración (véase la Figura 5.3). Ajustaremos los parámetros siguientes:

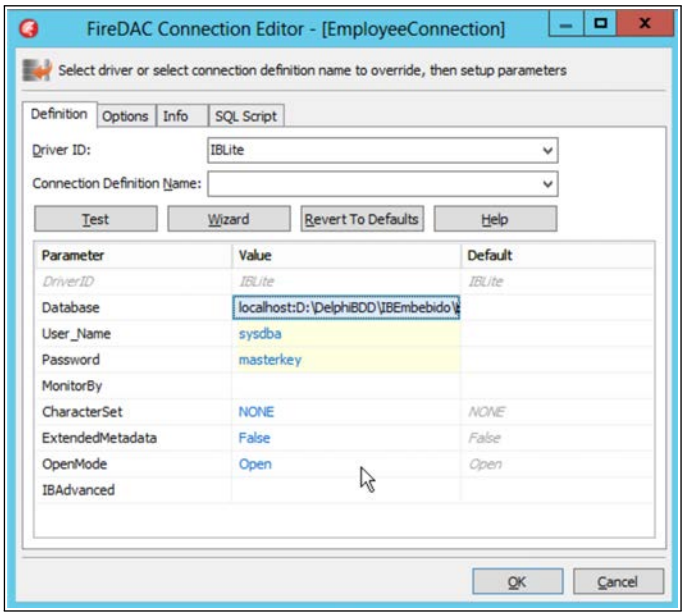


Figura 5.3 CONFIGURACIÓN DE CONEXIÓN A LA BASE DE DATOS

- **Controlador**: Desplegamos la lista **DRIVER ID** y seleccionamos el elemento **IBLITE**, estableciendo así el controlador adecuado para trabajar con esta base de datos.

- **Base de datos**: En el campo **DATABASE** introducimos la ruta[3] y nombre de la base de datos `employee.gdb`. Podemos copiar dicha base de datos a la carpeta de nuestro proyecto, a fin de simplificar el posterior despliegue a otros dispositivos. Esta ruta nos permitirá trabajar durante la fase de diseño, pero será necesario adecuarla posteriormente.

- **Credenciales**: Introducimos el valor `sysdba` en el apartado **USER_NAME** y el valor `masterkey` en el campo **PASSWORD**. Estas son las credenciales por defecto de InterBase. También daremos el valor `False` a la propiedad `LoginPrompt` del `TFDQuery` para que no se soliciten las credenciales al intentar conectar.

Finalizada la configuración, podemos usar el botón **TEST** de la ventana anterior o bien dar el valor `True` a la propiedad `Connected` del `TFDQuery`, comprobando así que la conexión es posible.

> **NOTA**
>
> *En este proyecto no vamos a utilizar un archivo de definición de conexiones, sino que almacenaremos todos los parámetros en el propio `TFDConnection` con el objetivo de poder ajustarlos en ejecución, tal y como se explicará más adelante.*

5.4.2 Selección de datos

El objeto de la aplicación será facilitar la consulta de departamentos de una empresa, obteniendo la lista de empleados de cada departamento y, por cada empleado, su historial de cambios de salario. Por tanto tendremos que introducir en el módulo de datos tres componentes `TFDQuery`. Estos se vincularán automáticamente el anterior `TFDConnection`. A continuación usaremos el editor de consultas para configurar los datos a obtener por cada uno:

[3] Es importante preceder la ruta con `localhost:` a fin de que la conexión mediante IBLite funcione desde el IDE de Delphi.

```
1 SELECT * FROM DEPARTMENT
```

Listado 5.1 Consulta para el componente `DepartmentTable`

```
1 SELECT * FROM EMPLOYEE
2 WHERE DEPT_NO = :DEPT_NO
```

Listado 5.2 Consulta para el componente `EmployeeTable`

```
1 SELECT * FROM SALARY_HISTORY
2 WHERE EMP_NO = :EMP_NO
```

Listado 5.3 Consulta para el componente `SalaryHistoryTable`

Como puede apreciarse, `EMPLOYEE` es la tabla de detalle respecto a los datos de `DEPARTMENT`, mientras que `SALARY_HISTORY` es la tabla de detalle respecto a la primera. Tenemos, por tanto, dos relaciones maestro/detalle en cascada. Para terminar de configurarlas deberemos añadir dos componentes `TDataSource` al módulo de datos, enlazando cada uno con una tabla maestra. A continuación usaremos la propiedad `MasterSource` de las dos tablas de detalle fijando el vínculo.

Para terminar con la configuración de datos, introduciremos en el modulo un `TFDGUIxWaitCursor` y un `TDFPhysIBDriverLink`. El módulo quedará finalmente como se muestra en la Figura 5.4. Podemos dar el valor `True` a la propiedad `Active` de los tres `TFDQuery` a fin de ver los datos en el formulario mientras lo diseñamos.

5.4.3 Diseño de la interfaz

Volvamos al formulario vacío con que contaba el proyecto inicialmente. Lo primero que haremos será agregar una referencia al módulo de datos. Como ya sabemos podemos hacerlo mediante la opción **FILE—USE UNIT**.

Dado que pretendemos que la aplicación se utilice en cualquier tipo de dispositivo, incluidos los que no contemplan el uso de múltiples ventanas, basaremos la interfaz de usuario un único formulario estructurado en tres secciones distintas, usando para ello un control `TTabControl`. Tras agregar este elemento al formulario reproduciremos los pasos indicados a continuación:

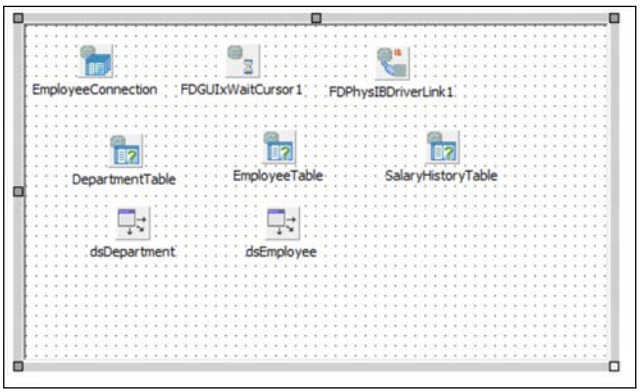


Figura 5.4 COMPONENTES FIREDAC AGREGADOS AL MÓDULO DE DATOS

1. Damos el valor `Client` a la propiedad `Align` del `TabControl` para que ocupe todo el espacio disponible. Elegimos la opción `None` para la propiedad `TabPosition` a fin de ocultar las pestañas de las páginas.

2. Abrimos el menú contextual del `TTabControl` y elegimos la opción **ADD TTABITEM** para agregar una nueva página. Esta ocupará el mismo espacio asignado al `TTabControl`.

3. Introducimos en el `TTabItem` un `TToolBar`. El valor de la propiedad `Align` tomará automáticamente el valor `Top`. Insertamos en ese componente un `TLabel` y asignamos el valor `Departamentos` a su propiedad `Text`.

4. Añadimos un `TListView` al mismo `TTabItem`, dándole el valor `Client` a su propiedad `Align`. Daremos a este componente el nombre (propiedad `Name`) `ListViewDEPARTMENT`.

5. Repetimos los pasos 2 a 4 agregando dos `TTabItem` adicionales con la misma estructura: un `TToolBar`, que además del `TLabel` también contendrán un `TButton`, y un `TListView`.

6. Configuraremos los `TButton`, que quedarán situados en el extremo izquierdo de sus respectivos contenedores `TToolBar`, asignando el valor

`backtoolbutton` a sus propiedades `StyleLookup`[4] y asignando el texto `Volver` a sus propiedades `Text`, respectivamente.

La estructura general de la interfaz de usuario, y las relaciones entre contenedores y controles contenidos, será la mostrada en la Figura 5.5. En ella pueden verse también los nombres asignados a los distintos `TListView`.

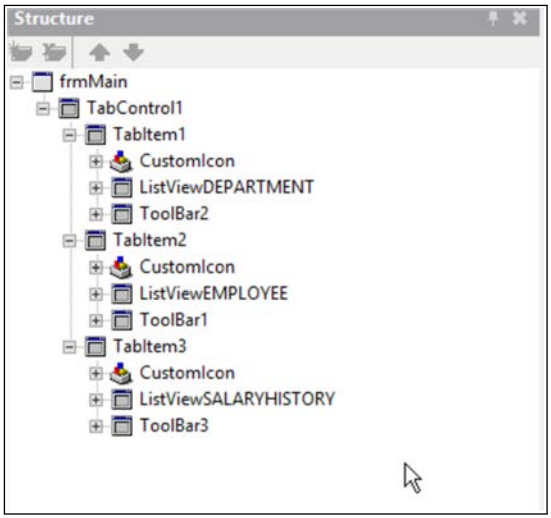


Figura 5.5 ESTRUCTURA DE LA INTERFAZ DE USUARIO

5.4.4 Enlace con los datos

El siguiente paso será vincular cada uno de los `TListView` con su origen de datos. También los `TLabel` incluidos en la parte superior del segundo y tercer `TTabItem` estarán vinculados a un campo de datos.

Para empezar tendremos que agregar al formulario tres `TBindSourceDB`, usando su propiedad `DataSet` para vincularlos a los componentes `TFDQuery`

[4]Esta propiedad tiene asociada una lista desplegable de la que podemos elegir el estilo que se le quiera dar al botón.

que tenemos en el módulo de datos. De esta forma ya podremos acceder a sus datos desde el **LIVEBINDINGS DESIGNER**, panel que abriremos con la opción **BIND VISUALLY** de cualquiera de los `TListView`.

Encontrándonos en el diseñador visual de *LiveBindings*, mediante operaciones de arrastras y soltar estableceremos las conexiones mostradas en la Figura 5.6. La propiedad `Item.Text` de los `TListView` corresponde al texto que se mostraría en cada elemento de la lista. Con la propiedad `ItemHeader.Text` se introduce una cabecera de separación entre elementos, cuyo título sería el contenido de la columna enlazada.

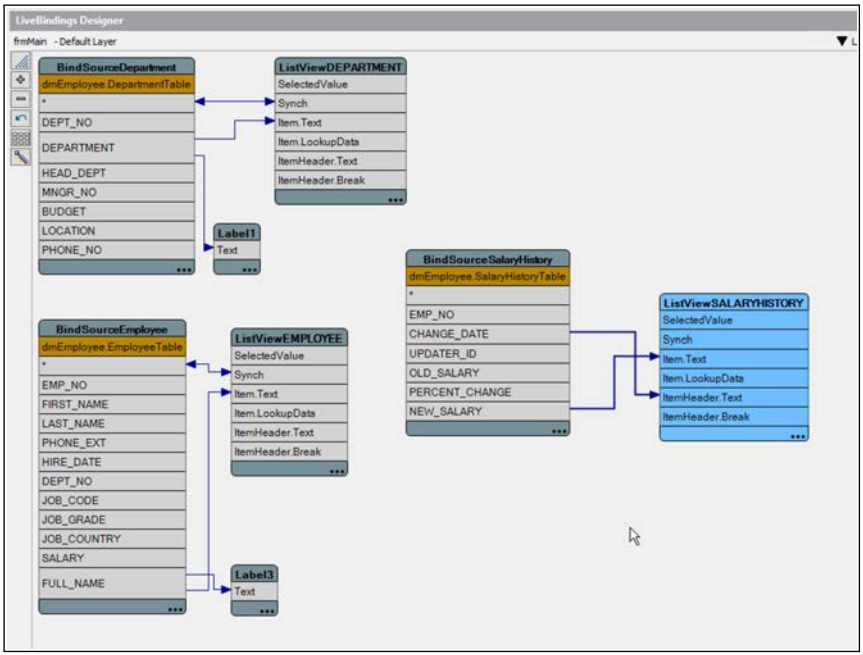


Figura 5.6 CONFIGURACIÓN DE ENLACE A DATOS DEL FORMULARIO

Con esta configuración conseguimos que el primer `TListView` muestre el nombre de cada departamento, el segundo el nombre de los empleados que corresponden al mismo y el tercero el historial de cambios de salario de un empleado. La relación maestro/detalle entre estas tablas se estableció explícitamente

en el módulo de datos, mediante el uso de parámetros en las consultas y la configuración de la propiedad `MasterDataSource`, por lo que en la interfaz no es necesario especificar nada adicional.

5.4.5 Gestión de los eventos

La página inicialmente activa del `TTabControl` será la primera, con la lista de departamentos, siendo necesario un mecanismo que permita abrir la lista de empleados del departamento elegido y, una vez en ella, dar paso al historial de cambios de salario del empleado que se seleccione. Además tendremos que facilitar la vuelta atrás de cada paso.

Tendremos que controlar, por tanto, la pulsación sobre un elemento de los dos primeros `TListView` y la pulsación sobre los botones de vuelta atrás. Podríamos gestionar estas tareas definiendo acciones, con un `TActionList`, que pudiesen reutilizarse a través de otros mecanismos, por ejemplo atajos de teclado, pero por simplicidad nos limitaremos a asociar el código de cada tarea directamente al gestor de eventos apropiado.

Los eventos a controlar son `OnItemClick` en los `TListView` y `OnClick` en los `TButton`. En ambos casos el cambio de una página a otra se efectúa con una simple asignación a la propiedad `ActiveTab` del `TTabControl`, como se muestra en el siguiente listado:

```
1  procedure TfrmMain.ListViewDEPARTMENTItemClick(const Sender:↙
       ↳ TObject;
2    const AItem: TListViewItem);
3  begin
4    TabControl1.ActiveTab := TabItem2;
5  end;
6
7  procedure TfrmMain.Button1Click(Sender: TObject);
8  begin
9    TabControl1.ActiveTab := TabItem1;
10 end;
11
12 procedure TfrmMain.ListViewEMPLOYEEItemClick(const Sender: ↙
       ↳ TObject;
```

```
13    const AItem: TListViewItem);
14  begin
15    TabControl1.ActiveTab := TabItem3;
16  end;
17
18  procedure TfrmMain.Button2Click(Sender: TObject);
19  begin
20    TabControl1.ActiveTab := TabItem2;
21  end;
```

Listado 5.4 Eventos de navegación por la interfaz

5.4.6 Comprobación de la aplicación

En este momento el proyecto está preparado para que comprobemos su funcionalidad, en el mismo equipo en que ha sido desarrollado y habiendo activado el acceso a datos durante la fase de diseño, estableciendo la propiedad `Active` de los `TFDQuery`. No tenemos más que pulsar **F9** y verificar que se muestra la lista de departamentos obtenida de la base de datos InterBase, que al seleccionar uno de ellos aparece la lista de empleados y que es posible acceder al historial de cambios de salario de estos, tal y como se muestra en la Figura 5.7.

Si estas pruebas son correctas podemos pasar a la siguiente fase, preparando el proyecto para su despliegue en otras plataformas. En caso contrario tendríamos que revisar[5] toda la configuración de componentes y código para detectar el problema que pudiese existir.

5.5 Configuración de despliegue

La configuración actual de nuestro proyecto le permite funcionar en el mismo equipo en que se ha desarrollado, pero no se comportaría correctamente si simplemente llevásemos el ejecutable a otro dispositivo, especialmente en el caso

[5]Recuerda que tienes el proyecto completo a tu disposición en el repositorio de código asociado al libro.

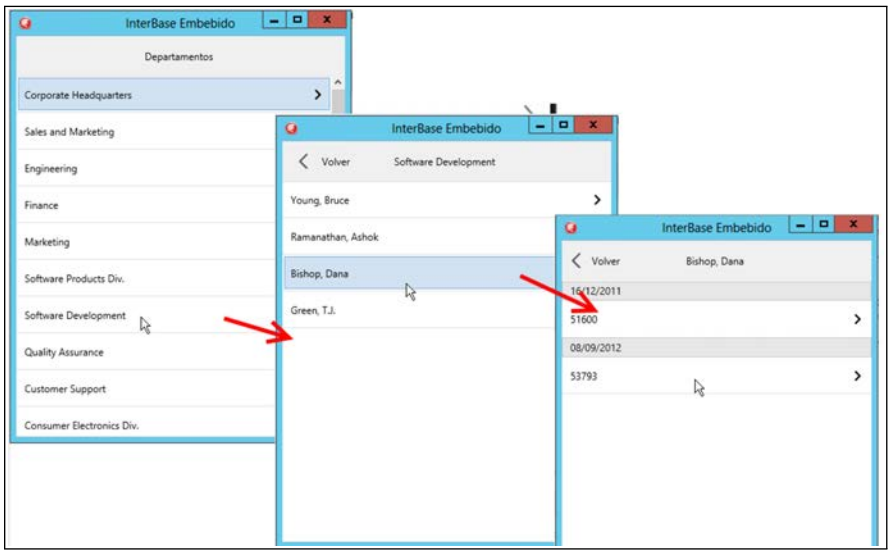


Figura 5.7 EL PROGRAMA EN FUNCIONAMIENTO EN WINDOWS

de iOS y Android. Es preciso introducir algunos cambios en el programa, por una parte, y ajustar la configuración de despliegue para cada sistema, por otra. Ese es el objetivo de esta última sección del capítulo.

5.5.1 Ruta de la base de datos

El primer cambio que hemos de introducir afecta a la configuración del componente `TFDConnection`, concretamente a la ruta en la que supuestamente ha de encontrarse el archivo `employee.gdb` conteniendo la base de datos. Dicha ruta existe en nuestro ordenador, pero no estará disponible en un dispositivo iOS o Android ya que en estos está restringido el acceso de las aplicaciones a las ubicaciones de almacenamiento. Además el controlador que estamos utilizando para acceder a la base de datos también ha de ser distribuido con la aplicación, así como la correspondiente licencia.

Ocupémonos por ahora de la ubicación en que se encontrará la base de datos, ajustando la configuración del `TFDConnection` adecuadamente. La base de datos se copiará a la carpeta de documentos asociada a la aplicación, una ubicación que podemos obtener mediante el método `GetDocumentsPath` de la clase `TPath`. A dicha ruta agregaremos el nombre de la base de datos, usando el separador que corresponda al sistema en que se ejecute la aplicación. Con este fin usaríamos el método `Combine` de la misma clase.

Nos interesa establecer la ruta correcta de la base de datos justo antes de que el `TFDConnection` abra la conexión, algo que ocurrirá en cuanto se ejecute la aplicación ya que dimos el valor `True` a la propiedad `Active` de los tres `TFDQuery`. Por ello localizaremos el evento `OnBeforeConnect` del `TFDConnection`, introduciendo el código mostrado a continuación:

```
1 procedure TdmEmployee.EmployeeConnectionBeforeConnect(Sender↙
      ↘ : TObject);
2 begin
3   {$IF DEFINED(iOS) or DEFINED(ANDROID)}
4    EmployeeConnection.Params.Values['Database'] :=
5       TPath.Combine(TPath.GetDocumentsPath, 'EMPLOYEE.GDB');
6   {$ENDIF}
7 end;
```

Listado 5.5 Establecemos la ubicación de la base de datos antes de conectar

Mediante la propiedad Params del componente `TFDConnection` accedemos a los parámetros de configuración, modificando el parámetro `Database`. Hemos usado directivas de compilación condicional a fin de que este código únicamente se ejecute para sistemas Android e iOS. Para los demás podríamos seguir utilizando un archivo de configuración con la cadena de conexión, como hacíamos en capítulos previos, o bien incluir la base de datos[6] en la misma carpeta de la aplicación y dejar la ruta simplemente como el nombre de la base de datos.

[6]Algunos sistemas operativos utilizan sistemas de archivos en los que se diferencia entre mayúsculas y minúsculas, por lo que es importante que el valor asignado a `Database` coincida exactamente con el nombre del archivo, incluyendo mayúsculas y minúsculas.

5.5.2 Redistribución de IBLite y su licencia

A continuación tendremos que incluir en el despliegue del proyecto tanto los archivos redistribuibles de IBLite como la licencia que permitirá que este funcione. Para ello usaremos la opción **PROJECT—DEPLOYMENT**, abriendo el panel de opciones de despliegue. Este cuenta en su parte superior con una barra de herramientas entre las que encontraremos el botón **ADD FEATURED FILES**, encargado de abrir la ventana mostrada en la Figura 5.8. En ella marcaremos la opción **INTERBASE TOGO** para todas las plataformas o bien para aquellas en las que vayamos a distribuir la aplicación. En este caso concreto, dado que hemos utilizado IBLite en el proyecto, es importante que desmarquemos la distribución del archivo `reg_ibtogo.txt` y marquemos en su lugar `reg_iblite.txt`. Estos archivos contienen la licencia de uso de InterBase ToGo e IBLite, respectivamente.

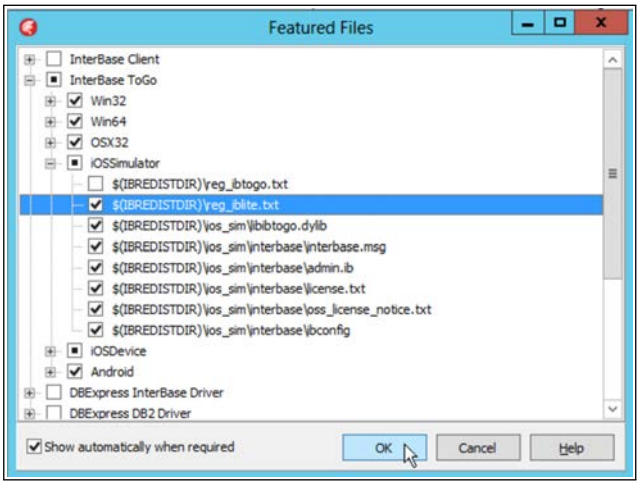


Figura 5.8 COMPONENTES REDISTRIBUIBLES A AGREGAR AL PROYECTO

Al hacer clic en el botón **OK** comprobaremos cómo los archivos de IBLite se agregan a cada una de las configuraciones. En la lista que hay en la parte superior derecha del panel seleccionaremos una plataforma y, si lo deseamos, una configuración concreta, verificando los elementos a desplegar junto con la aplicación.

5.5.3 Distribución de la base de datos

Además del controlador IBLite y su licencia, al desplegar la aplicación también tendremos que copiar la propia base de datos `EMPLOYEE.GDB` en la ubicación adecuada. La configuración para ello se establece desde el mismo panel en que estamos trabajando. Con el segundo de los botones agregaremos el archivo a la lista. A continuación lo seleccionaremos y usaremos el botón mostrado en la Figura 5.9 para establecer la ruta en la que se copiaría en el destino. Esta última acción hemos de efectuarla una vez por cada plataforma de despliegue, ya que las ubicaciones varían según el sistema operativo de destino.

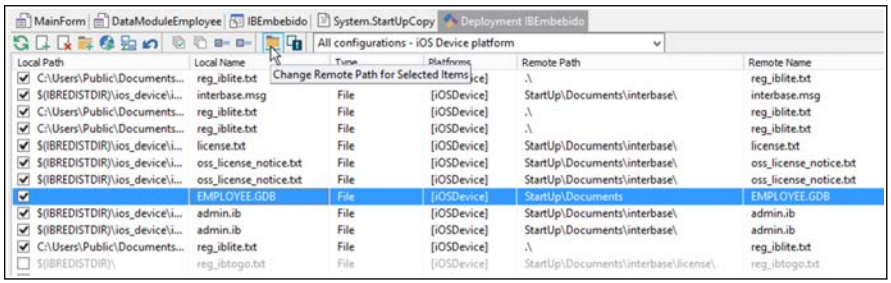


Figura 5.9 INCLUIMOS LA PROPIA BASE DE DATOS ENTRE LOS ARCHIVOS A DISTRIBUIR

En sistemas iOS la ubicación de destino ha de ser `StartUp\Documents`, mientras que en Android dicha ruta sería `assets\internal`. Si fuésemos a desplegar la aplicación en OS X o Win32/Win64 escogeríamos la ubicación para la que hubiésemos configurado la aplicación.

5.5.4 Configuración de permisos

En sistemas Android una aplicación que va a utilizar IBLite o InterBase ToGo precisa solicitar al usuario, en el momento de la instalación, permisos para leer y escribir en almacenamiento externo donde está alojada la base de datos, los permisos se denominan **READ EXTERNAL STORAGE** y **WRITE EXTERNAL STORAGE**, así como para comunicarse con el gestor de datos (permiso **INTER-**

NET). Para activar esos permisos abriremos la ventana de opciones del proyecto, iremos a la página **USES PERMISSIONS** y marcaremos los citados permisos, como puede verse en la Figura 5.10.

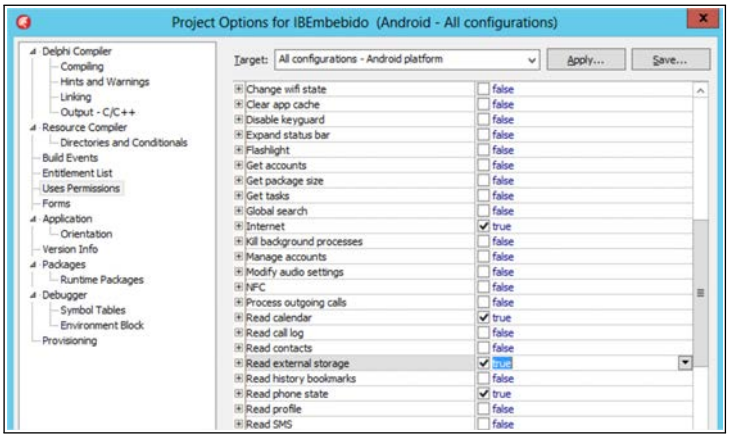


Figura 5.10 CONFIGURACIÓN DE PERMISOS DE LA APLICACIÓN

> **NOTA**
>
> *Para ampliar información sobre configuraciones de permisos, opciones de despliegue, el uso del Asistente para despliegue o PAServer y muchos otros detalles relacionados con el desarrollo para dispositivos móviles con Delphi recomendamos el texto* ***Desarrollo de aplicaciones iOS/Android con Delphi****, de la misma editorial y autor.*

5.5.5 Comprobación de la aplicación

Habiendo finalizado la configuración de despliegue, estamos en disposición de probar la aplicación en cualquier otro sistema. Para ello seleccionaremos la plataforma objetivo en el Gestor de proyectos (véase la Figura 5.11). La rama **ANDROID** nos permite probar tanto en el emulador de Android incluido con Delphi como en dispositivos reales, siempre que estos estén conectados por USB

al equipo de desarrollo y apropiadamente configurados. Para desplegar en las demás plataformas deberemos instalar el software PAServer correspondiente. En el caso de iOS es posible usar un dispositivo físico, conectado a un Mac, o bien un emulador de iOS, para lo cual será preciso instalar el SDK de iOS para OS X y XCode.

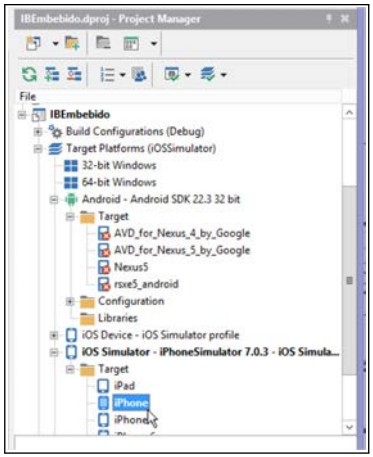


Figura 5.11 ELEGIMOS LA PLATAFORMA EN QUE SE DESPLEGARÁ EL PROYECTO

Asumiendo que quisiésemos probar en el emulador de iOS, iniciaríamos el **PASERVER MANAGER** en OS X y a continuación, usando su menú desplegable (véase la Figura 5.12) pondríamos en marcha un servidor. Este quedaría a la escucha, esperando la comunicación desde el entorno de Delphi.

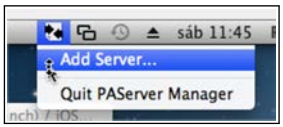


Figura 5.12 PASERVER MANAGER

Seleccionamos como plataforma de destino el emulador de iOS y ejecutamos el proyecto. Si es la primera vez que desplegamos sobre iOS, aparecerá un asistente que nos permitirá establecer la configuración de comunicación: dirección

IP o nombre del equipo con OS X y puerto en que está a la escucha PAServer. Con estos datos se procederá a transferir todos los archivos necesarios al Mac, encargándose PAServer de iniciar el emulador de iOS y lanzar la ejecución del proyecto.

El programa, como puede apreciarse en la Figura 5.13, ofrece exactamente la misma funcionalidad que ya habíamos verificado en el equipo de desarrollo, si bien su apariencia se ajusta al estilo de iOS.

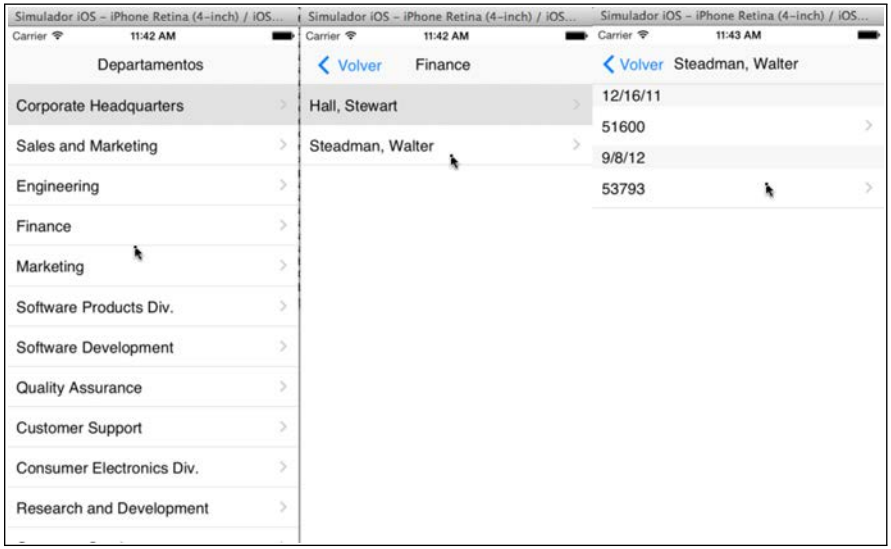


Figura 5.13 LA APLICACIÓN FUNCIONANDO EN EL EMULADOR DE IOS

En caso de que la aplicación no llegarse a mostrar la interfaz de usuario, cerrándose automáticamente al intentar ejecutarla, lo más probable es que exista un problema con la conexión a los datos. Lo más recomendable es efectuar la conexión, dando el valor `True` a la propiedad Activede los `TFDQuery`, en el evento `OnCreate` del formulario, protegiendo esas sentencias para capturar cualquier excepción que se produzca y mostrarla en pantalla.

Si, por ejemplo, no hubiésemos copiado el archivo de licencia de IBLite al dispositivo, o estuviese en la localización incorrecta, al ejecutar la aplicación

aparecería un error como el de la Figura 5.14. En estos casos podemos examinar el directorio `scratchdir` que utiliza PAServer (véase la Figura 5.15) para copiar los archivos que después se despliegan al emulador, verificando que están todos los que deberían y en las ubicaciones correctas.

Figura 5.14 ERROR AL INTENTAR CONECTAR

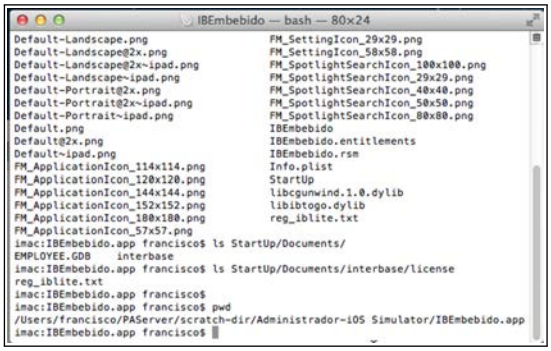


Figura 5.15 COMPROBAMOS LOS ARCHIVOS EN LA CARPETA DE DESPLIEGUE

5.6 Resumen

En este capítulo hemos comprobado cómo podemos utilizar exactamente los mismos componentes FireDAC que utilizábamos para SQLite o Access para operar sobre bases de datos InterBase, concretamente la edición embebida IBLite que puede desplegarse en dispositivos móviles junto con nuestra aplicación. De esta forma seguimos contando con un RDBMS completo incluso cuando nuestros programas se ejecutan en iOS o Android. También hemos aprendido a configurar el proyecto para distribuir los componentes de IBLite y su licencia de uso, todo ello a través del desarrollo completo de un ejercicio.

Aunque IBLite e InterBase ToGo también pueden utilizarse en equipos de escritorio, en esta configuración posiblemente nos interese más recurrir al servidor local de InterBase. También es posible que nos interese usar otros tipos de bases de datos que son usuales cuando se almacena la información directamente en ordenadores personales, en lugar de en servidores de datos o dispositivos móviles. Este será el tema que se aborde en el siguiente capítulo.

Capítulo 6

DELPHI Y BASES DE DATOS DE ESCRITORIO

Al desarrollar aplicaciones que van a utilizarse en ordenadores personales, ya sean de escritorio o portátiles, es muy probable que surja la necesidad de operar con orígenes de datos exclusivos de estas plataformas. Los usuarios de ordenadores personales tienden a almacenar su información en el propio ordenador, no en servidores de datos remotos, usando para ello formatos que, en general, son específicos de cada sistema operativo.

En Windows, por ejemplo, es habitual que se utilicen bases de datos Microsoft Access y hojas de cálculo Microsoft Excel para guardar la información. En muchos casos Excel se emplea como una base de datos simple, permitiendo que

personas que no tienen experiencia con bases de datos puedan estructurar su información de manera más sencilla. También es habitual que los datos se alojen directamente en archivos de texto, sencillamente usando comas u otro símbolo como separadores de campos y disponiendo cada entrada en una nueva fila.

El objetivo de este capítulo es que aprendamos a utilizar algunos de estos orígenes de datos, usados fundamentalmente en el escritorio. Ya sabemos cómo conectar con bases de datos Access, lo hicimos en uno de los ejercicios de capítulos previos, por lo que en las siguientes secciones nos ocuparemos de las hojas de cálculo Access y los archivos de texto.

6.1 Acceso a datos en archivos de Microsoft Excel

Microsoft Excel es una herramienta muy popular en distintos sistemas operativos, siendo muchos los profesionales que lo utilizan no solamente para efectuar cálculos o elaborar gráficas, sino también como almacenes de datos. La estructura de un archivo Excel se presta a este uso, ya que cada hoja puede ser tratada como una tabla de una base de datos, estructurada en filas y columnas alojando un dato en cada cruce.

No obstante Excel no es un gestor de datos, por lo que las operaciones que es posible realizar sobre el contenido de las hojas no son las habituales del lenguaje SQL. Se trata de una aplicación orientada al uso de fórmulas en las que existen relaciones arbitrarias entre las celdillas, no aplicándose las restricciones a las que estamos habituados en una base de datos.

6.1.1 FireDAC y Excel

A la hora de trabajar con información alojada en archivos Excel tenemos fundamentalmente dos alternativas: usar COM (*Component Object Model*) para acceder al modelo de objetos Excel y controlarlo desde nuestra aplicación, lo cual implica que Excel ha de estar instalado en el ordenador donde vaya a utilizarse nuestra aplicación, o bien acceder al archivo en sí como si de una base de datos se tratase.

Entre los controladores de FireDAC no encontraremos uno específico que nos permita abrir un archivo Excel y acceder a su contenido. Por esta razón es necesario recurrir a un acceso indirecto, utilizando para ello el controlador ODBC para Excel ofrecido por Microsoft. Podemos usar dicho controlador sin necesidad de tener instalado Excel, únicamente habría que instalar en los equipos donde vaya a utilizarse el propio controlador. Este puede descargarse desde Microsoft sin ningún coste y está disponible para distintas versiones de Windows.

> **NOTA**
>
> *Según indica Microsoft, fabricante del controlador ODBC para Excel, las operaciones que es posible llevar a cabo con dicho controlador son limitadas. Básicamente solo se contemplan las consultas (`SELECT`) y la adición de nuevos datos (`INSERT`), pero no el resto de operaciones (`UPDATE` o `DELETE`). Puedes obtener más información buscando el documento `INF: Excel ODBC Driver and Text ODBC Driver Notes` en soporte de Microsoft.*

6.1.2 Configuración de FireDAC para usar un controlador ODBC

Para utilizar un controlador ODBC es necesario facilitar una cadena de parámetros de configuración, especificando cuál es el controlador a usar, el nombre de usuario y contraseña si procede, la ruta y nombre de la base de datos, etc. Opcionalmente pueden existir parámetros específicos de cada controlador. Toda esta información la introduciríamos, directa o indirectamente, en la propiedad `Params` del componente `TFDConnection`.

En el caso concreto del controlador ODBC para Excel, es importante que se incluyan tres parámetros adicionales: `Mode`, `ReadOnly` y `HDR`. Con los dos primeros se establece el modo de apertura del archivo Excel. Habitualmente les asignaremos los valores `ReadWrite` y `false`, indicando así que no se abrirá solo para lectura. En cuanto al tercer parámetro, también de tipo booleano, su finalidad es indicar si las hojas del archivo Excel cuentan con una primera fila conteniendo los nombres de las columnas o no, según lo cual le asignaremos el valor `true` o `false`, respectivamente. Por defecto dicho parámetro toma

el valor `false`, asumiendo que no existe una cabecera con los nombres de los campos.

Mediante el parámetro `Database` facilitaremos al controlador la ubicación y nombre del archivo Excel con el que queremos trabajar. Dependiendo de la versión del controlador se permitirá el acceso únicamente al antiguo formato `.xls` o también al más reciente `.xlsx`.

La configuración completa, tal y como la introduciríamos en la citada propiedad `Params`, sería la mostrada en el Listado 6.1.

```
1 DriverID=ODBC
2 ODBCDriver={Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, ↙
      ↳ *.xlsb)}
3 Database=Ruta\NombreArchivo.xls
4 User_Name=admin
5 ODBCAdvanced=Mode=ReadWrite;ReadOnly=false; HDR=yes
```

Listado 6.1 Parámetros de configuración ODBC

6.1.3 Consultas sobre un libro Excel

Una vez establecida la conexión con el archivo Excel, el libro que contiene es visto por el controlador como una base de datos. Cada una de sus hojas sería como una tabla, pero hemos de tener en cuenta que su nombre siempre irá seguido del sufijo `$`. Dado que este es un carácter especial, para usarlo es preciso introducir el nombre entre corchetes. Suponiendo que en el archivo existiese una hoja llamada `Hoja1`, una consulta como la mostrada a continuación recuperaría todo su contenido.

```
1 SELECT *
2 FROM [Hoja1$]
```

Listado 6.2 Consulta para obtener el contenido de una hoja

La selección de columnas de la hoja se efectuará directamente en caso de que cuente con una cabecera con los nombres y se haya usado el parámetro

HDR=yes en la conexión, como si de una tabla corriente se tratase. En caso contrario se recurriría a incluir en la propia tabla el rango de celdillas a tomar, usando la misma sintaxis habitual en Excel. Por ejemplo:

```
1 SELECT *
2 FROM [Hoja1$A1:B5]
```

Listado 6.3 Selección de un rango de celdillas

En este ejemplo se recuperaría el contenido de cinco filas, 1 a 5, con dos columnas, A y B. También es posible usar únicamente el nombre de columna para recuperar todas las filas que contenga. Si la hoja tuviese un encabezado y se hubiese incluido el parámetro HDR=yes, asumiendo que el nombre de las columnas fuese Continente y Pais podríamos recuperar la misma información así:

```
1 SELECT Continente, Pais
2 FROM [Hoja1$]
```

Listado 6.4 Selección de un rango de celdillas

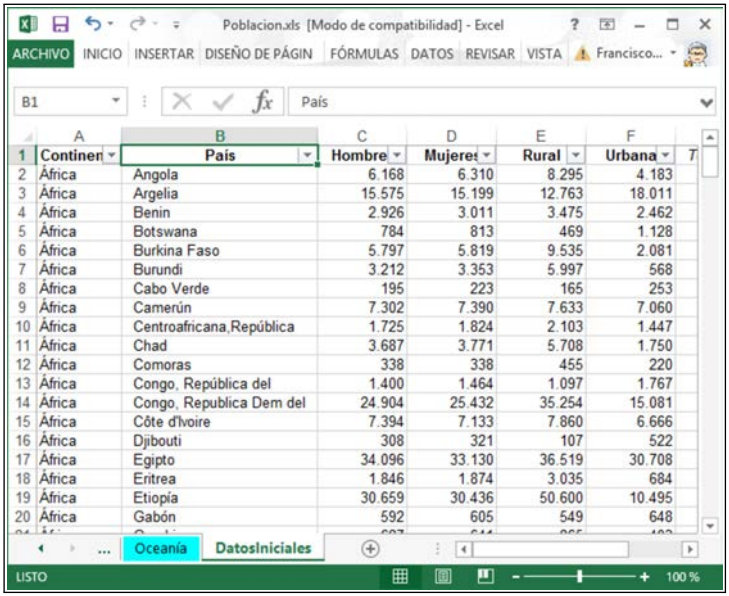
En las consultas es posible utilizar las cláusulas WHERE, ORDER BY, GROUP BY, INNER JOIN, etc., así como funciones de agregado de datos. Si el controlador lo permite, también podemos realizar otras operaciones como la inserción de nuevos datos

6.1.4 Supuesto práctico

Comprobemos con un ejercicio práctico cómo usaríamos la información almacenada en un archivo Excel y cómo agregar nuevos datos al mismo, todo ello desde un programa Delphi ejecutándose en Windows.

Partiremos de una hoja de cálculo Excel en la que tenemos una página, llamada DatosIniciales, con una lista de datos relativos a la población por países de todo el mundo con datos segmentados por sexo y zona, tal y como se muestra en la Figura 6.1. Cada país podría aparecer varias veces, conteniendo cada fila datos parciales de la población. El objetivo es obtener esos

datos en nuestra aplicación, agregarlos de forma que obtengamos la población de los países de un continente concreto y, finalmente, agregar esa información en nueva hoja al archivo Excel.



	A	B	C	D	E	F
1	Continen	País	Hombre	Mujeres	Rural	Urbana
2	África	Angola	6.168	6.310	8.295	4.183
3	África	Argelia	15.575	15.199	12.763	18.011
4	África	Benin	2.926	3.011	3.475	2.462
5	África	Botswana	784	813	469	1.128
6	África	Burkina Faso	5.797	5.819	9.535	2.081
7	África	Burundi	3.212	3.353	5.997	568
8	África	Cabo Verde	195	223	165	253
9	África	Camerún	7.302	7.390	7.633	7.060
10	África	Centroafricana,República	1.725	1.824	2.103	1.447
11	África	Chad	3.687	3.771	5.708	1.750
12	África	Comoras	338	338	455	220
13	África	Congo, República del	1.400	1.464	1.097	1.767
14	África	Congo, Republica Dem del	24.904	25.432	35.254	15.081
15	África	Côte d'Ivoire	7.394	7.133	7.860	6.666
16	África	Djibouti	308	321	107	522
17	África	Egipto	34.096	33.130	36.519	30.708
18	África	Eritrea	1.846	1.874	3.035	684
19	África	Etiopía	30.659	30.436	50.600	10.495
20	África	Gabón	592	605	549	648

Figura 6.1 CONTENIDO DE LA HOJA EXCEL

Como puede apreciarse en la Figura 6.1, la hoja cuenta con una primera fila que actúa como cabecera, asignando un nombre a cada una de las columnas. Esto nos permitirá referirnos a ellas mediante esos nombres en lugar de como A, B, etc.

Iniciaremos un nuevo proyecto en Delphi, seleccionando la plantilla en blanco de la categoría **MULTI-DEVICE APPLICATION**. Como es habitual, agregaremos al proyecto un módulo de datos y usaremos la opción **FILE—USE UNIT** para hacer referencia al mismo desde el formulario, separando así los componentes de acceso a datos de la interfaz de usuario del programa.

6.1.5 Configuración de la conexión por ODBC

Introducimos en el módulo de datos un componente `TFDConnection` y hacemos doble clic sobre él para abrir el cuadro de diálogo de configuración (véase la Figura 6.2).

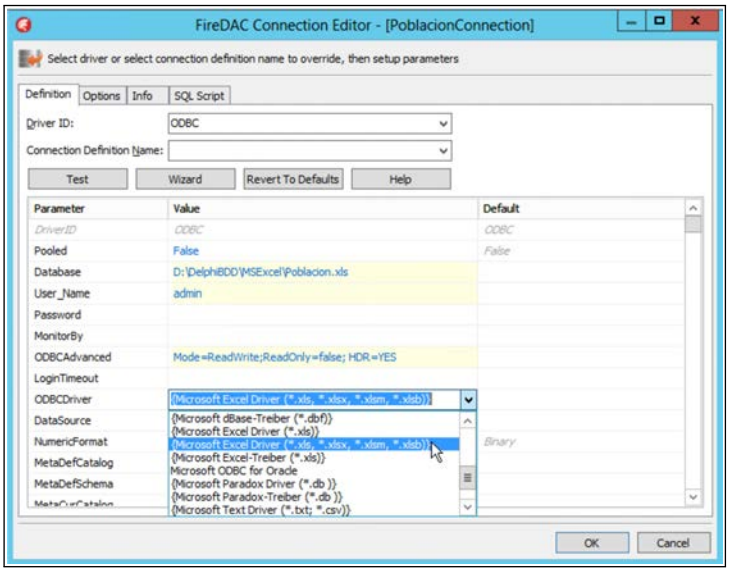


Figura 6.2 CONFIGURACIÓN DE LA CONEXIÓN AL ARCHIVO EXCEL

EJEMPLO 6.1 `MSExcel.dproj`

Puedes encontrar este proyecto completo en la carpeta `MSExcel`. Además del proyecto también se incluye el archivo Excel con los datos. Para poder ejecutar el programa deberás instalar el controlador ODBC para Microsoft Excel, si no está en tu sistema, y modificar el parámetro `Database` para establecer la ruta donde está el archivo.

Seleccionamos de la lista **DRIVER ID** el controlador `ODBC`, abriendo a continuación la lista asociada a la propiedad `ODBCDriver` para elegir el controlador correspondiente a Excel. En la propiedad `Database` facilitamos la ubicación y nombre del archivo. Finalmente, en la propiedad `ODBCAdvanced` establecemos la configuración avanzada descrita anteriormente, fjando el modo de apertura de la hoja Excel e indicando que existe una cabecera con los nombres de las columnas.

Finalizada la configuración, podemos usar el botón **TEST** para comprobar que es posible abrir el archivo Excel, así como introducir alguna consulta de prueba en la página **SQL SCRIPT**, ajustándonos a la sintaxis que se explicó previamente.

6.1.6 Consulta de recuperación de datos

El paso siguiente será añadir al módulo de datos un componente `TFDQuery`, abrir el editor asociado e introducir la consulta SQL que queremos ejecutar. Nos interesa obtener la población de cada país por sexo para todos los países de un continente. Dado que cada país puede aparecer más de una vez, usaremos la función de agregado `SUM` y la cláusula `GROUP BY`, tal y como se muestra a continuación:

```
1 SELECT Pais, SUM(Hombres) AS TotalHombres, SUM(Mujeres) AS ↙
    ↳ TotalMujeres
2 FROM [DatosIniciales$]
3 WHERE Continente = 'Europa'
4 GROUP BY Pais
```

Listado 6.5 Titulo

Tras escribir la consulta el botón **EXECUTE** la ejecutará y nos permitirá comprobar tanto los datos devueltos como la estructura de estos. En este caso, como se muestra en la Figura 6.3, recuperamos tres columnas, una de tipo cadena y dos números en punto flotante. Usando la página **OPCIONES** de la ventana de parámetros de conexión (Figura 6.2) podríamos, en caso necesario, modificar la asociación entre tipos de datos Excel y tipos Delphi.

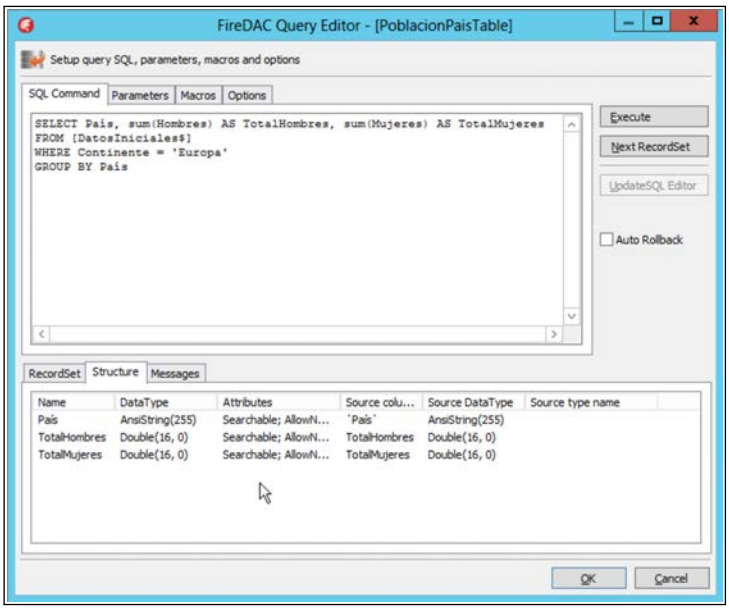


Figura 6.3 CONSULTA PARA OBTENER LOS DATOS DE LA HOJA

Para terminar con el módulo de datos, agregaremos el habitual componente `TFDGUIxWaitCursor` y un `TFDPhysODBCDriverLink`. Opcionalmente podemos dar el valor `True` a la propiedad `Active` del `TFDQuery` para ver los datos durante el diseño de la interfaz.

6.1.7 Diseño de la interfaz de usuario

Disponiendo de los elementos que nos permitirán acceder al contenido del archivo Excel, procedamos ahora a diseñar una sencilla interfaz de usuario que los muestre. Para ello introduciremos en el formulario los elementos siguientes:

- Un control `TToolBar` ajustado a la parte superior del formulario.
- Un `TButton` que quedará contenido en el anterior. Cambiaremos su propiedad `Text` asignándole el texto `Exportar`.

- Un control `TGrid` a cuya propiedad `Align` daremos el valor `Client`, a fin de que ocupe el resto del espacio disponible en el formulario.

Usando el **LIVE BINDINGS DESIGNER** vincularemos el control `TGrid` con el componente `TFDQuery`, a fin de mostrar el contenido de las tres columnas resultantes de la consulta. Los enlaces son los mostrados en la Figura 6.4.

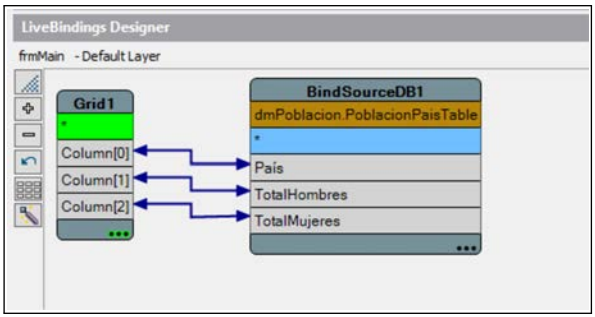


Figura 6.4 ENLACE ENTRE LA CUADRÍCULA Y LOS DATOS

El control `Grid` cuenta con un editor de columnas, accesible mediante la opción **COLUMNS EDITOR** de su menú contextual, desde el que podemos personalizar la visualización de las columnas, por ejemplo asignándoles una cabecera más adecuada en lugar de utilizar los nombres de los campos (véase Figura 6.5).

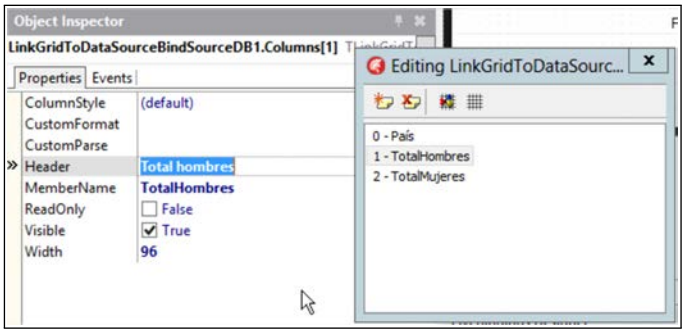


Figura 6.5 CONFIGURACIÓN DE LAS COLUMNAS DEL `TGRID`

Solo con esto ya deberíamos poder ver la información devuelta por la consulta en la cuadrícula de datos. Podemos comprobarlo dándole temporalmente el valor `True` a la propiedad `Active` del `TFDQuery`. El resultado debería ser el que aparece en la Figura 6.6.

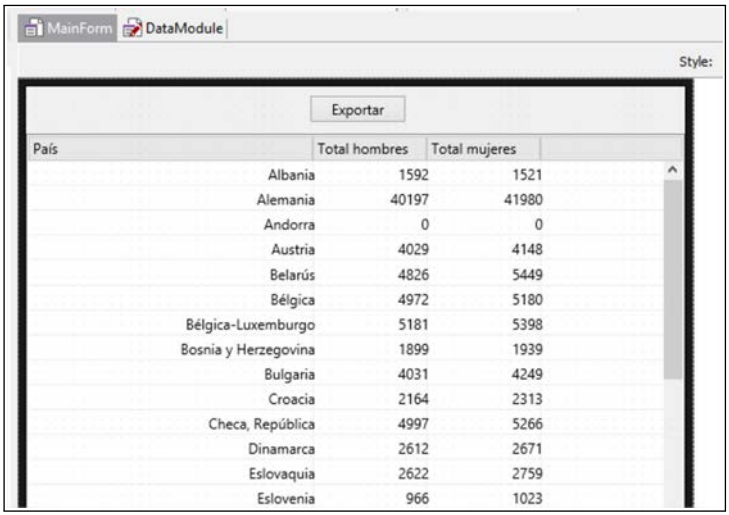


Figura 6.6 LA CUADRÍCULA MOSTRANDO LOS DATOS DE LA HOJA EXCEL

6.1.8 Apertura de la conexión

Un archivo Excel no es una base de datos, razón por la que no puede ser abierto simultáneamente para escritura por dos o más programas. Este es un factor importante a tener en cuenta. Si tenemos la hoja abierta en Excel nuestro programa fallará al ejecutarse. Incluso si tenemos activa la conexión en el diseñador, al intentar ejecutar el proyecto se producirá una excepción. Por ello daremos el valor `False` a la propiedad `Active` del `TFDQuery`, controlando la conexión desde el código del programa.

Concretamente usaremos el evento `OnCreate` del módulo de datos para activar la conexión. Hacemos doble clic sobre el módulo de datos e introducimos una simple sentencia de asignación, como se muestra en el listado siguiente:

```
procedure TdmPoblacion.DataModuleCreate(Sender: TObject);
begin
  PoblacionPaisTable.Active := True;
end;
```

Listado 6.6 Apertura de la conexión desde el módulo de datos

Dado este paso deberíamos poder ejecutar el programa y navegar por los datos sin ningún problema.

6.1.9 Exportación de los datos

La última operación que nos falta por implementar es la exportación de los datos que estamos viendo en el programa, añadiéndolos como una nueva hoja al archivo Excel. Esta será la finalidad del botón dispuesto en la parte superior, por lo que hacemos doble clic sobre él y le asociamos el código facilitado en el Listado 6.7.

Utilizamos el método `ExecSQL` del componente `TFDConnection` para ejecutar directamente las sentencias SQL apropiadas. Concretamente son dos: una para crear la nueva hoja en el libro Excel y otra para introducir en ella los datos que queremos escribir. La primera de las consultas fallará en caso de que en el libro ya exista una hoja con el mismo nombre. Podríamos controlar dicha excepción y modificar el nombre, por ejemplo agregándole un número detrás.

```
procedure TfrmMain.Button1Click(Sender: TObject);
var
  query: String;
begin
  with dmPoblacion.PoblacionConnection do
  begin
    query := 'CREATE TABLE SumEuropa (Pais varchar, Hombres ↙
      ↳ integer, Mujeres integer)';
    ExecSQL(query);

    query :=
      'INSERT INTO [SumEuropa$](Pais, Hombres, Mujeres) ' +
```

```
12          'SELECT Pais, sum(Hombres) AS TotalHombres, sum(
                ↳ Mujeres) AS TotalMujeres ' +
13          'FROM [DatosIniciales$] ' +
14          'WHERE Continente = ''Europa'' ' +
15          'GROUP BY Pais';
16      ExecSQL(query);
17    end;
18  end;
```

Listado 6.7 Código asociado al botón de exportación

Ahora ya estamos en disposición de ejecutar el programa. Este mostraría (véase la Figura 6.7) la cuadrícula con la población por países, según se especifico en la consulta de recuperación de datos.

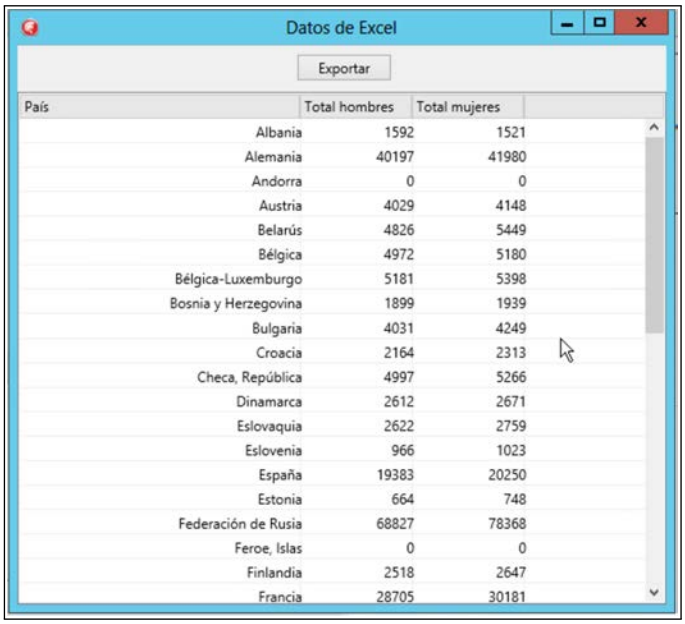
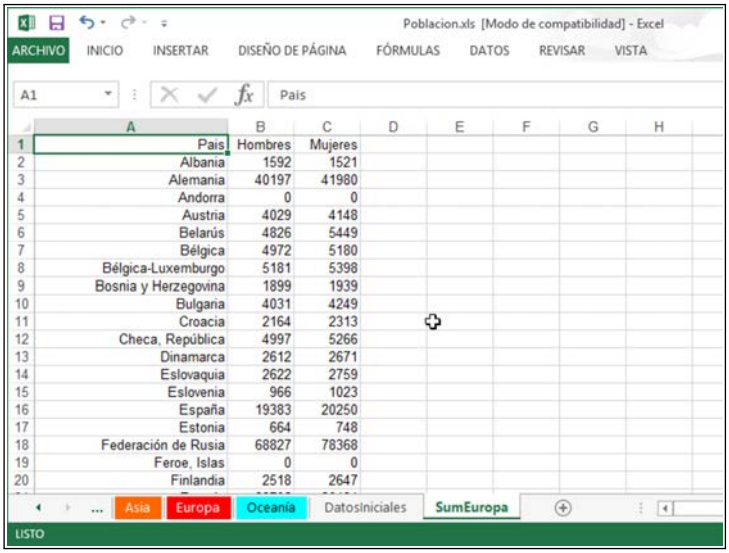


Figura 6.7 LA APLICACIÓN EN FUNCIONAMIENTO

Tras hacer clic en el botón **EXPORTAR** podemos abrir la hoja de cálculo en Excel y verificar que, tal y como se muestra en la Figura 6.8, existe una nueva hoja con los datos introducidos desde la aplicación Delphi.



	A	B	C
1	Pais	Hombres	Mujeres
2	Albania	1592	1521
3	Alemania	40197	41980
4	Andorra	0	0
5	Austria	4029	4148
6	Belarús	4826	5449
7	Bélgica	4972	5180
8	Bélgica-Luxemburgo	5181	5398
9	Bosnia y Herzegovina	1899	1939
10	Bulgaria	4031	4249
11	Croacia	2164	2313
12	Checa, República	4997	5266
13	Dinamarca	2612	2671
14	Eslovaquia	2622	2759
15	Eslovenia	966	1023
16	España	19383	20250
17	Estonia	664	748
18	Federación de Rusia	68827	78368
19	Feroe, Islas	0	0
20	Finlandia	2518	2647

Figura 6.8 LA HOJA DE DATOS AGREGADA POR LA APLICACIÓN DELPHI AL LIBRO EXCEL

NOTA

La técnica descrita en esta sección, usando controladores ODBC, también nos permitirá acceder a datos alojados en formatos antiguos pero aún presentes en muchas empresas, como dBase o Paradox. Solamente tendríamos que elegir el controlador ODBC apropiado.

6.2 Uso de datos almacenados en archivos de texto

En muchas ocasiones los datos que ha de utilizar una aplicación se encuentran almacenados en archivos de texto simples. Estos pueden haber sido creados y ser utilizados directamente por el usuario, con un editor sencillo, o bien ser generados por aplicaciones que utilizan formatos más especializados. En cualquier caso nos interesará saber cómo podemos ejecutar consultas sobre esos datos para utilizarlos en un proyecto propio.

Los archivos de texto se estructuran en líneas, al igual que una hoja de cálculo lo hace en filas o una base de datos en registros. Por regla general existe un carácter de retorno de carro que marca el final de cada línea. Para diferenciar los campos existentes en cada línea suele utilizarse un separador concreto, como pueden ser tabuladores, comas o puntos y comas[1]. Dicho separador no debe aparecer en el interior de un dato. Para evitar confusiones, es posible entrecomillar aquellos campos que sean cadenas de texto. También es importante determinar cuál será el separador de miles si existen datos numéricos con parte no entera.

Aunque podríamos utilizar un controlador ODBC u OLEDB para acceder al contenido de un archivo de texto, la operaciones que podemos efectuar sobre ellos son, en general, limitadas. En esta sección aprenderemos a conectar una interfaz de usuario Delphi con el contenido de un archivo CSV de forma directa, sin usar más que algunos componentes FireDAC.

6.2.1 Lectura de archivos CSV con FireDAC

Lo primero que necesitamos saber es cómo podemos leer el contenido de un archivo de texto, por ejemplo en formato CSV, manteniendo la información en un componente sobre el que podamos realizar consultas y otras operaciones.

FireDAC nos ofrece un mecanismo genérico, representado por el componente `TFDBatchMove`, para leer datos de un origen y escribirlos en un des-

[1] Uno de los formatos de archivos de texto más conocido es el CSV (*Comma Separated Values*), denominado así porque cada dato se separa del siguiente por una coma.

tino. Tanto origen como destino pueden ser archivos de texto, componentes que alojan conjuntos de datos (como un `TFDQuery`) o conexiones con RDBMS. En la página **FIREDACT ETL** de la Paleta de componentes (véase la Figura 6.9) encontraremos tanto el componente `TFDBatchMove` como los componentes encargados de leer y escribir los datos para cada categoría. Así tenemos un `TFDBatchMoveTextReader` y un `TFDBatchMoveTextWriter`, para trabajar sobre archivos de texto; un `TFDBatchMoveDataSetReader` y un `TFDBatchMoveDataSetWriter`, para leer y escribir de componentes que representan conjuntos de datos, y finalmente un `TFDBatchMoveSQLReader` y un `TFDBatchMoveSQLWriter`, encargados de la lectura y escritura en conexiones con bases de datos.

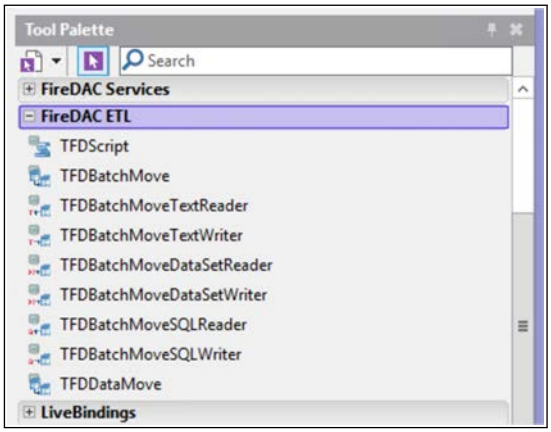


Figura 6.9 COMPONENTES FIREDAC PARA TRANSFERIR DATOS ENTRE ORIGEN Y DESTINO

Cada uno de estos componente contendría la configuración del origen o el destino de la transferencia de datos, encargándose el `TFDBatchMove` de realizar las conversiones necesarias para trasladar la información de un punto a otro. El componente `TFDBatchMove` dispone de las propiedades `Reader` y `Writer`, a las que se conectaría el objeto que actuará de lector y el que operará como escritor, respectivamente. La transferencia se completará mediante una invocación al método `Execute`.

La configuración general para llevar a cabo un proceso de transferencia de datos con `TFDBatchMove` es el representado esquemáticamente en la Figura 6.10. El citado componente se comunica bidireccionalmente con el lector y el escritor. Estos usan su configuración específica para llevar a cabo la lectura, en este ejemplo de un archivo CSV, y la escritura, por ejemplo en un `TFDQuery`.

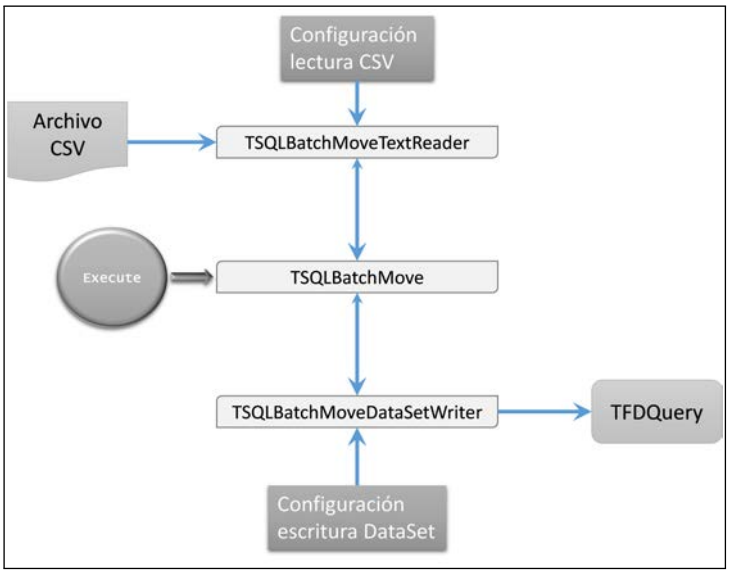


Figura 6.10 PROCESO DE TRANSFERENCIA DE DATOS CON `TBatchMove`

6.2.2 Procesamiento local de consultas SQL

Para conectar una interfaz de usuario Delphi a un origen de datos necesitamos que la información esté alojada en algún componente derivado de `TDataSet`, como puede ser `TFDQuery`. Estos componentes se vinculan a un componente de conexión, un `TFDConnection`, que en último término es el encargado de enviar las consultas al gestor de datos para que se ejecuten.

En los capítulos previos hemos trabajado con Microsoft Access, SQLite e InterBase, todos ellos con capacidad para evaluar y ejecutar sentencias SQL.

Incluso en el caso de Excel, según comprobamos en la sección previa de este mismo capítulo, el controlador ODBC es capaz de interpretar un subconjunto bastante amplio de SQL. Al trabajar con información procedente de un archivo de texto, sin embargo, no existe ningún software en el que los componentes FireDAC deleguen esa tarea, por lo que es preciso contar con algún mecanismo que se encargue del procesamiento de las sentencias SQL.

FireDAC puede usar el motor de SQLite para procesar consultas SQL sin necesidad de conectar con base de datos alguna. Para ello se utilizan dos componentes:

- `TFDConnection`: Se configura para que utilice el controlador de SQLite, pero sin asignar valor alguno a la propiedad `Database`. De esta manera no se creará ningún archivo en disco.

- `TFDLocalSQL`: Conectado al anterior, este componente se encarga de procesar localmente las consultas SQL recibidas, utilizando para ello el motor de SQLite.

Cualquier componente conectado al `TFDConnection` podrá utilizar la capacidad de procesamiento local del `TFDLocalSQL`. Por ejemplo, al conectar un `TFDQuery` y ejecutar una consulta será el `TFDLocalSQL` el que la interprete, determinando de dónde se extraerá la información, cómo se filtrarán las filas y proyectarán las columnas, etc.

La información sobre la que trabajarán los componentes `TFDLocalSQL` y `TFDConnection` puede tener una representación estática, estableciéndose en la fase de diseño cuáles son los componentes en que reside. Para ello se utilizaría la propiedad `DataSets` del `TFDLocalSQL`. También es posible generar la información dinámicamente, por ejemplo a través de un `TFDBatchMove`, respondiendo a los eventos del componente `TFDLocalSQL`.

El evento `OnGetDataSet` del componente `TFDLocalSQL` se genera si, al procesar una consulta, se hace referencia a un conjunto de datos (una tabla o una vista) no disponible en la propiedad `DataSets`. El método asociado a dicho evento aporta el nombre del conjunto de datos, que puede ser utilizado para prepararlo y devolverlo como resultado. Los parámetros `ADataSet` y `AOwned`, recibidos por referencia en dicho evento, permiten asignar el conjunto de datos e indicar si este ha de ser gestionado por el `TFDLocalSQL`.

6.2.3 Consultas sobre archivos CSV

Comprobemos, mediante el desarrollo de un caso práctico, cómo podemos efectuar consultas sobre los datos contenidos en un archivo CSV usando los componentes enumerados en los apartados previos.

> **EJEMPLO 6.2** `DatosCSV.dproj`
>
> Puedes encontrar este proyecto completo en la carpeta `DatosCSV`. Además del proyecto también se incluyen dos archivos CSV con datos: `iris.csv` y `cars.csv`. Tanto estos como cualquier otro archivo CSV pueden ser usados en las consultas, siempre que se alojen en la carpeta apropiada. Tendrás que modificar la ubicación de esta en el método asociado al evento `OnGetDataSet` del componente `TFDLocalSQL` (en el módulo de datos).

Partiremos, como es habitual, creando un nuevo proyecto vacío de tipo **MULTI-DEVICE APPLICATION** y agregando un módulo de datos al mismo. No debemos olvidar agregar al formulario una referencia al módulo de datos. En los apartados siguientes se describen los pasos que hemos de reproducir.

6.2.4 Componentes de conexión y procesamiento de consultas

Comencemos introduciendo en el módulo de datos los componentes de conexión y procesamiento de las consultas. Serán tres componentes: un `TFDConnection`, un `TFDLocalSQL` y un `TFDQuery`. Si los añadimos en ese orden, los dos últimos se enlazarán automáticamente con el primero.

Del `TFDConnection` cambiaremos la propiedad `DriverName`, seleccionando `SQLite` de la lista de valores posibles, y daremos el valor `False` a la propiedad `LoginPrompt`. Con esta configuración el `TFDConnection` delegará la ejecución de las consultas SQL en el `TFDLocalSQL` conectado a él. De los otros dos componentes solamente estableceríamos la propiedad

`Connection`, en caso de que se no se hubiese creado automáticamente el enlace con el `TFDConnection`.

También agregaremos al módulo un componente `TFDGUIxWaitCursor` y un `TFDPhysSQLiteDriverLink`, introduciendo así en el código las referencias necesarias para incorporar el controlador de SQLite.

6.2.5 Lectura de los datos CSV

Nuestro objetivo es conectar la interfaz de usuario al componente TFDQuery, permitiendo ejecutar distintas consultas a través de él obteniendo datos de archivos CSV. Sin embargo, si en este momento introdujésemos una consulta en la propiedad `SQL` del `TFDQuery` y le diéramos el valor `True` a su propiedad `Active` el resultado sería un error. El `TFDConnection` no está realmente conectado a una base de datos y el `TFDLocalSQL` no encontraría la tabla solicitada.

Para leer los datos del archivo CSV adecuado, entregándoselos al componente `TFDLocalSQL` que, a su vez, los facilitaría al `TFDQuery`, tendremos que añadir varios componentes más al módulo de datos, tal y como se aprecia en la Figura 6.11. Vamos a ir abordando su configuración paso a paso.

El primer componente a insertar es el `TFDBatchMove`, encargado de gestionar el proceso de lectura del archivo CSV y la transferencia de los datos a un derivado de `TDataSet`. Al inicio de dicho proceso es posible llevar a cabo una exploración de las primeras filas de datos a fin de determinar su estructura. Este comportamiento lo controla la propiedad `Analyze` que, de no estar vacía, invocará al método `GuessFormat` de `TFDBatchMove` para deducir si existe o no una cabecera, el separador de campos, etc., dependiendo de las opciones que se hayan activado (véase la Figura 6.12).

> **NOTA**
>
> *El método `GuessFormat` únicamente se utiliza cuando la transferencia de datos implica a un origen que no contiene metadatos, como es el caso de los archivos de datos. Si se usase un origen con metadatos estos describirían la estructura de la información, no necesitándose el proceso mencionado.*

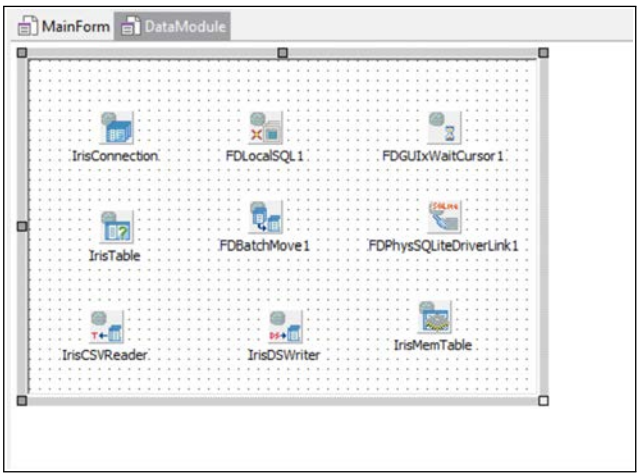


Figura 6.11 COMPONENTES A INTRODUCIR EN EL MÓDULO DE DATOS

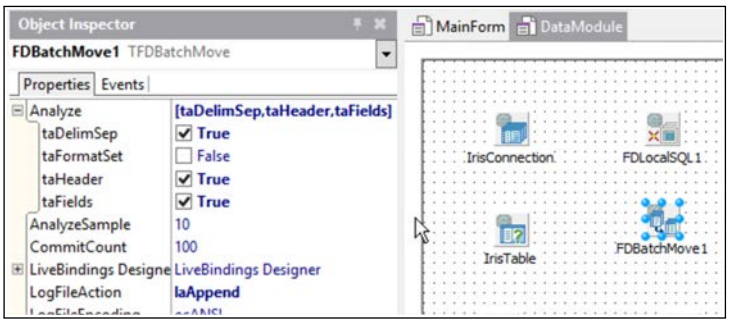


Figura 6.12 CONFIGURACIÓN DEL `TFDBatchMove`

A continuación añadiremos un componente `TFDBatchMoveTextReader`. Este se enlazará automáticamente con el anterior. Si conocemos la estructura de los archivos con los que vamos a trabajar, en este caso se trata de archivos CSV, podemos usar las propiedades `Separator`, `Delimiter` y `FormatSettings`

para configurar el proceso de lectura[2]. Esta última propiedad determina cuáles serán los separadores decimales, de fechas, de miles, símbolos de moneda y otros parámetros.

En nuestro caso indicaremos que el separador de campos, asignado a la propiedad `Delimiter` (véase la Figura 6.13), es la coma y que el separador de parte decimal, propiedad `FormatSettings.DecimalSeparator`, es el punto. Si fuésemos a usar siempre el mismo archivo CSV podríamos asignar su ubicación y nombre a la propiedad `FileName` durante la fase de diseño, pero en nuestro ejercicio ese es un parámetro que no conoceremos hasta que vaya a ejecutarse una consulta.

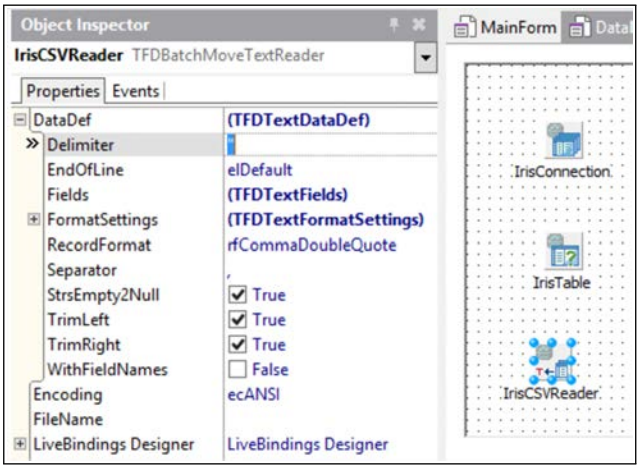


Figura 6.13 CONFIGURACIÓN DEL COMPONENTE `TFDBatchMoveTextReader`

El paso siguiente será introducir en el módulo de datos un componente `TFD-BatchMoveDataSetWriter`, que se encargará de guardar los datos leídos del archivo CSV en un conjunto de datos, y un componente `TFDMemTable`, cuyo objeto será servir como almacén temporal de esos datos, durante la transferencia.

[2]El componente `TFDBatchMoveTextReader` es genérico, no específico para archivos CSV, pudiendo leer prácticamente cualquier archivo de texto que use unos separadores concretos para líneas, campos, etc.

En este momento tenemos en el módulo de datos todos los elementos que necesitamos para poder leer el contenido de una archivo CSV, transferirlo al `TFDMemTable` y, desde ahí, entregarlo al `TFDLocalSQL` para que lo introduzca en el `TFDQuery`. La secuencia de eventos sería la siguiente:

1. Desde la interfaz del programa se introduce una consulta en la propiedad `SQL` del `TFDQuery` y se llama al método `Open` o se da el valor `True` a la propiedad `Active`.
2. El `TFDQuery` recurre al `TFDConnection` al que apunta su propiedad `Connection` para ejecutar la consulta y obtener el conjunto de datos resultante.
3. La consulta pasa del `TFDConnection` al motor de procesamiento local del componente `TFDLocalSQL`. Este no dispone de la tabla referenciada en la consulta en su propiedad `DataSets`, por lo que genera el evento `OnGetDataSet`.
4. En el método asociado a dicho evento introduciremos el código para leer el archivo CSV y transferir los datos al `TFDMemTable`, usando para ello los componentes que hemos agregado antes.
5. Asignaremos al parámetro `ADataSet` que se recibe en dicho método el contenido del `TFDMemTable`, conteniendo ya la información del archivo CSV.
6. El componente `TFDLocalSQL` entregará el resultado de la consulta de vuelta al `TDDQuery` que, a su vez, la facilitará a los controles de la interfaz de usuario.

El código indicado en el paso 4 será el mostrado en el Listado 6.8. El parámetro `AName` es el nombre de la tabla que aparece en la consulta, nombre que usaremos para abrir el archivo CSV y ejecutar la conversión.

```
1 procedure TdmIris.FDLocalSQL1GetDataSet(ASender: TObject;
2   const ASchemaName, AName: string;
3   var ADataSet: TDataSet; var AOwned: Boolean);
4 begin
5   try
```

```
6       IrisMemTable.Close;
7       IrisCSVReader.FileName := 'DatosCSV\' + AName + '.csv';
8       FDBatchMove1.Execute;
9       ADataSet := IrisMemTable;
10      AOwned := True;
11    except on E: Exception do
12      ShowMessage(E.Message);
13    end;
14  end;
```

Listado 6.8 Transferencia de datos CSV al `TFDMemTable`

6.2.6 Diseño de la interfaz

Nuestra interfaz de usuario estará basada una vez más en un componente `TGrid` que, mediante *LiveBindings*, vincularemos con el `TFDQuery` del módulo de datos. Además también dispondremos un `TToolBar` en la parte superior que servirá de contenedor de un `TEdit` y un `TButton`. En el `TEdit` podemos introducir en la propiedad `Text` una consulta de ejemplo, como se muestra en la Figura 6.14.

Cada vez que se haga clic en el `TButton` cerraremos la consulta actual, visible en el `TGrid`, y procederemos a ejecutar la que se hubiese introducido en el `TEdit`. El código necesario para ello es el mostrado a continuación:

```
1   procedure TfrmMain.Button1Click(Sender: TObject);
2   begin
3     with dmIris.IrisTable do
4     begin
5       Close;
6       SQL.Clear;
7       SQL.Add(edQuery.Text);
8       Open;
9     end;
10  end;
```

Listado 6.9 Código asociado al botón que ejecutará la consulta

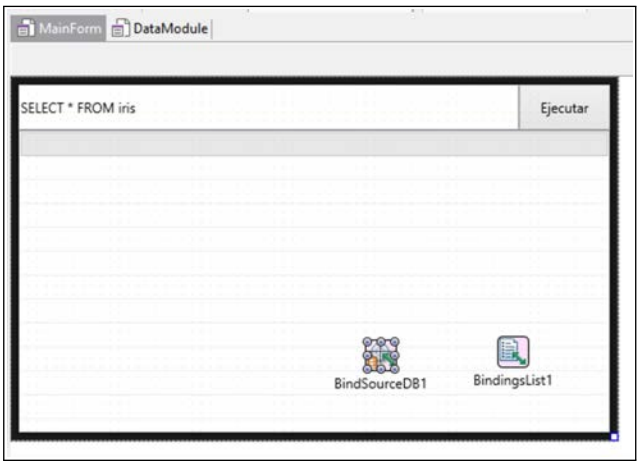


Figura 6.14 ASPECTO DE LA INTERFAZ EN EL DISEÑADOR DE FORMULARIOS

Con esto ya tenemos el programa terminado. Podemos ejecutarlo y probar distintas consultas (véase la Figura 6.15) sobre los archivos CSV de ejemplo o cualesquiera otros que dispongamos en la misma carpeta.

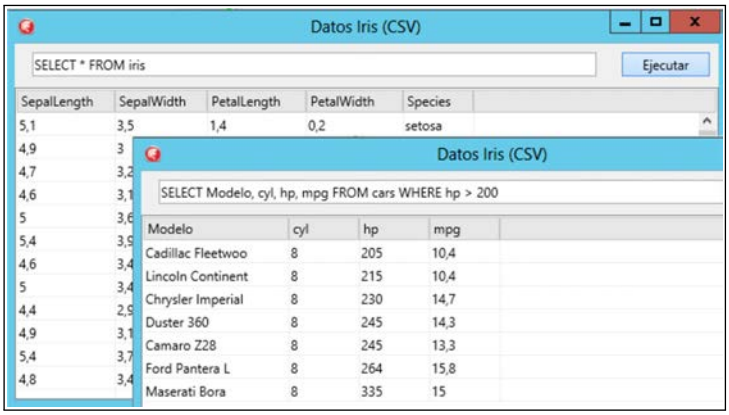


Figura 6.15 EL PROGRAMA MOSTRANDO DATOS DE DOS ARCHIVOS CSV

A pesar de que estemos trabajando con información alojada en un archivo de texto, las consultas pueden usar filtros, agrupaciones e incluso relaciones entre tablas, por ejemplo de tipo maestro/detalle en caso de que tuviésemos un archivo CSV con los datos maestros y otra con los de detalle.

NOTA

La información accesible mediante la interfaz de usuario de este programa podría modificarse, pero los cambios quedarían almacenados en el `TFDQuery` *y no llegarían a transferirse a los archivos CSV de origen. Para ello tendríamos que utilizar un conjunto adicional de componentes* `TFDBatchMove`*, con una configuración inversa a la que hemos empleado para la lectura. Al igual que la lectura, la escritura del contenido actual del* `TFDQuery` *podría realizarse a demanda, con un segundo botón que escribiese los cambios.*

6.3 Resumen

En este capítulo hemos aprendido a utilizar desde nuestras aplicaciones Delphi otros orígenes de datos de uso habitual en sistemas de escritorio, completando así el recorrido por las distintas configuraciones de acceso a datos de capítulos previos.

Primero se ha descrito el procedimiento a seguir para, mediante un controlador ODBC, acceder a información almacenada en hojas de cálculo Excel, así como para agregar a las mismas nueva información. Usando controladores ODBC podríamos también operar sobre otros tipos de bases de datos de escritorio, como dBase o Paradox, aunque estos formatos son cada vez menos habituales.

En la segunda parte del capítulo hemos entrado en contacto con un conjunto de componentes totalmente nuevo, como son los destinados a transferir datos entre distintas fuentes, la familia de componentes `TFDBatchMoveXXX`, y también los que facilitan el trabajo con datos cuando no existe una conexión real con un motor externo que los gestione, como `TFDLocalSQL` y `TFDMemTable`. Hemos aprendido a usar estos componentes con un objetivo concreto: usar en una aplicación Delphi información almacenada en archivos en formato CSV.

El próximo capítulo nos servirá para profundizar en el uso de algunos de los componentes introducidos en este. El objetivo será mostrar cómo una aplicación puede trabajar sobre datos que no se almacenan en archivos, sino que residen únicamente en memoria durante la ejecución.

Capítulo 7

BASES DE DATOS EN MEMORIA CON FIREDAC

No todas las aplicaciones que manejan datos durante su ejecución necesitan leerlos de un archivo u obtenerlos de una conexión a un RDBMS, ni posiblemente tampoco precisen almacenarlos al terminar la tarea. Hay muchas situaciones, especialmente cuando se utilizan dispositivos móviles, en las que los datos van generándose a lo largo de la ejecución, obtenidos de sensores integrados en el dispositivo, de una conexión a algún servicio u otro tipo de actividad.

En estos casos lo que necesita la aplicación es un almacenamiento temporal para la información, estableciendo su estructura al inicio y descartando los datos al terminar.

Esto no significa que no sea posible conservar estos datos para un uso posterior, en caso de que el usuario lo demandase. Para ello podrían utilizarse archivos en diferentes formatos, según las necesidades.

El objetivo de este capítulo es mostrarnos los procedimientos a seguir para desarrollar aplicaciones que trabajan con bases de datos en memoria, sin una conexión permanente a un gestor de datos, como podría ser InterBase o Access en el escritorio, y sin que sea imprescindible utilizar archivos en almacenamiento externo.

7.1 El motor local de SQL de FireDAC

FireDAC dispone de un componente llamado `TFDLocalSQL`, lo conocíamos en la segunda parte del anterior capítulo, que actúa como un motor local de procesamiento de consultas SQL. Conjuntamente con un `TFDConnection`, cuando el controlador elegido es `SQLite` y la propiedad `Database` se deja en blanco o se le asigna el valor especial `:memory:`, lo que obtenemos es una base de datos en memoria, tanto el gestor de almacenamiento de datos como el motor de procesamiento de consultas que actúa sobre él.

Al no estar conectados a una verdadera base de datos, o algún controlador que haga aparecer como tal un origen de otro tipo (como ocurre con ODBC y Excel), la meta-información que describe la estructura de la información no puede ser obtenida automáticamente. Básicamente tendremos tres alternativas para definir estas estructuras:

- Mediante consultas SQL de tipo DDL (*Data Definition Language*), usando sentencias del tipo `CREATE TABLE`.
- Usando las propiedades y métodos de algún derivado de `TFDDataSet` para definir sus columnas.
- Copiando esa información desde algún conjunto de datos al que tengamos acceso.

La base de datos en memoria puede mantener la meta-información de una base de datos completa, no solo de una tabla aislada. Es posible, por tanto,

definir varias tablas y establecer relaciones entre ellas, como haríamos con una base de datos estándar, así como definir vistas y otros objetos. Estos podrían conectarse después a componentes que representarían los distintos conjuntos de datos en memoria.

7.2 Conjuntos de datos en memoria

Otro de los componentes introducidos en el capítulo previo fue `TFDMemTable`, usado como almacenamiento temporal para realizar la conversión de datos CSV a un conjunto de datos. Dicho componente, como su propio nombre denota, ofrece la funcionalidad necesaria para mantener un conjunto de datos en memoria, con independencia de cómo haya sido generado.

La estructura del conjunto de datos alojado en un `TFDMemTable`, la lista de columnas que lo componen, sus nombres y tipos de datos, pueden ser definidos manualmente, usando para ello el Editor de columnas al que da paso la opción **FIELDS EDITOR** de su menú contextual (véase la Figura 7.1).

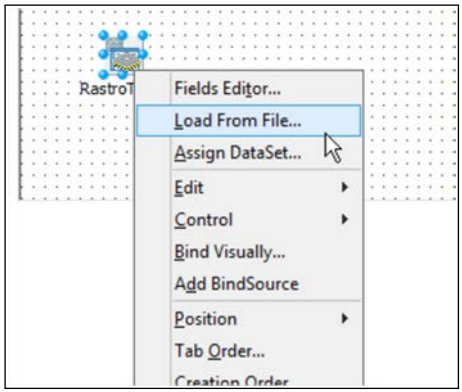


Figura 7.1 OPCIONES EN EL MENÚ CONTEXTUAL DE UN `TFDMEMTABLE`.

También es posible recuperar esa información de un archivo XML, con la opción **LOAD FROM FILE**, así como copiarla desde otro conjunto de datos que

tengamos en el mismo contenedor, mediante la opción **ASSIGN DATASET**. Una alternativa más sería la definición de las columnas mediante código.

Además de la estructura y los datos propiamente dichos, un `TFDMemTable` también incorpora la funcionalidad necesaria para conservar un registro de los cambios introducidos en los datos, pudiendo convertirlos en las sentencias de actualización apropiadas. En esencia, por tanto, tenemos todo lo necesario para trabajar sobre un conjunto de datos, con la excepción del mecanismo que permite obtenerlo de una fuente externa. Esto es lo que aportan componentes como `TFDTable`, `TFDQuery` y `TFDStoredProc`, que son tres versiones especializadas de `TFDMemTable` con capacidad para recuperar la información de estructura y contenido de una tabla, del resultado de una consulta SQL o del resultado de la ejecución de un procedimiento almacenado, respectivamente.

Los tres componentes citados usan internamente un `TFDMemTable`, por lo que la funcionalidad aportada por este está igualmente presente en todos ellos. Esto significa que a la hora de trabajar con datos en memoria podemos utilizar indistintamente uno u otro, según nuestras necesidades.

7.3 Definición de estructuras de datos en memoria

Para poder trabajar con una base de datos en memoria lo primero que tendremos que hacer será definir la estructura de cada una de las tablas que necesitemos. Con este fin podemos recurrir a diferentes opciones, según el tipo de componente que empleemos.

Si optamos por definir la estructura de las tablas durante la fase de diseño, que es la primera alternativa que va a describirse, habitualmente utilizaremos un componente `TFDMemTable` ya que este cuenta con opciones adicionales no presentes en `TFDQuery` o `TFDTable`, por ejemplo. El procedimiento básico, no obstante, es exactamente el mismo para los tres componentes.

La definición de la estructura de las tablas durante la ejecución, segundo enfoque que abordaremos más adelante, puede efectuarse con cualquiera de los componentes citados ya que, internamente, todos ellos tienen un `TFDMemTable`,

disponiendo en consecuencia de un conjunto común de propiedades y métodos que después `TFDQuery` y `TFDTable` extienden con miembros a medida.

7.3.1 En la fase de diseño

Los tres componentes antes citados ofrecen en su menú contextual una misma opción, **FIELDS EDITOR**, que da paso a un editor que ya conocemos de capítulos previos. En su momento lo usábamos para seleccionar las columnas de una tabla que queríamos incluir en una consulta. Ahora lo utilizaremos para definir las columnas de una tabla que aún no existe.

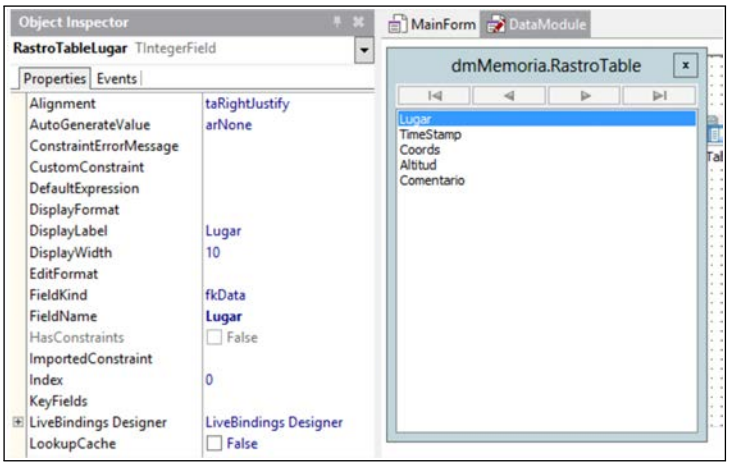


Figura 7.2 DEFINICIÓN DE COLUMNAS DE UNA TABLA CON EL EDITOR DE CAMPOS.

El procedimiento a seguir es siempre el mismo. Una vez abierto el editor de columnas reproducimos los pasos indicados a continuación:

1. Pulsamos **CONTROL+N** (o elegimos la opción **NEW FIELD** del menú contextual del editor) para agregar una nueva columna a la tabla.

2. Se abre un cuadro de diálogo, en el que especificaremos el nombre del campo, su tipo y su longitud.

3. Mediante el Inspector de objetos ajustamos cualquier otra característica del campo seleccionado en el editor (véase la Figura 7.2).

Una vez que hayamos definido las columnas del conjunto de datos que queremos usar, daremos el valor `True` a la propiedad `Active` del componente para crearlo. Esa acción creará el conjunto de datos en memoria, dejándolo preparado para su uso como cualquier otro conjunto de datos que hubiésemos obtenido de una base de datos.

Si el componente que estamos utilizando es un `TFDMemTable`, la creación del conjunto de datos se traducirá en la aparición de opciones adicionales en el menú contextual del componente. En la Figura 7.3, que muestra dicho menú, puede apreciarse la presencia de las opciones **CLEAR DATA** y **SAVE TO FILE**.

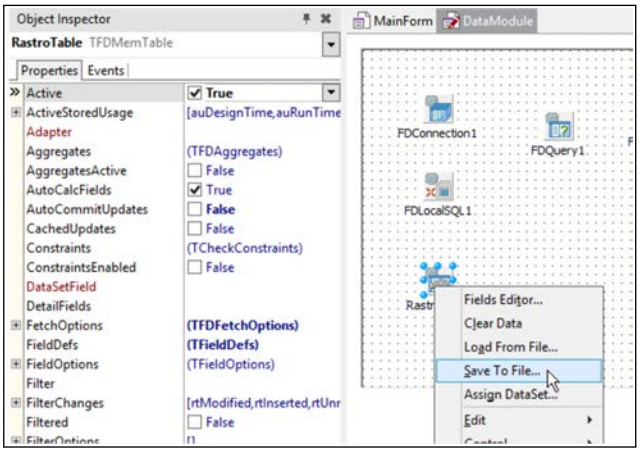


Figura 7.3 OPCIONES EN EL `TFDMEMTABLE` TRAS HABER GENERADO EL CONJUNTO DE DATOS

La primera opción eliminaría los datos contenidos en el componente, pero sin eliminar los datos de estructura. Con la segunda se almacenaría toda la información, tanto estructura de la tabla como su posible contenido, en un archivo XML. El formato de dicho archivo es el mostrado en el Listado 7.1. Su contenido podría ser recuperado con **LOAD FROM FILE**, ya sea del mismo componente o de otro distinto, pudiendo leerse durante la ejecución, con una

llamada al método `LoadFromFile`. Este es heredado por todos los derivados de `TFDDataSet`, entre los que se encuentran `TFDMemTable`, `TFDTable` y `TFDQuery`.

```
<?xml version="1.0" encoding="utf-8"?>
<FDBS Version="14">
 <Manager UpdatesRegistry="True">
  <TableList>
   <Table Name="RastroTable" SourceName="Table" TabID="0" ↙
       ↳ EnforceConstraints="False" MinimumCapacity="50" ↙
       ↳ CheckNotNull="False">
   <ColumnList>
    <Column Name="Lugar" SourceName="Lugar" SourceID="1" ↙
        ↳ DataType="Int32" Searchable="True" AllowNull="True↙
        ↳ " Base="True" OAllowNull="True" OInUpdate="True" ↙
        ↳ OInWhere="True" OriginColName="Lugar"/>
    <Column Name="TimeStamp" SourceName="TimeStamp" SourceID↙
        ↳ ="2" DataType="DateTime" Searchable="True" ↙
        ↳ AllowNull="True" Base="True" OAllowNull="True" ↙
        ↳ OInUpdate="True" OInWhere="True" OriginColName="↙
        ↳ TimeStamp"/>
    <Column Name="Coords" SourceName="Coords" SourceID="3" ↙
        ↳ DataType="AnsiString" Size="18" Searchable="True" ↙
        ↳ AllowNull="True" Base="True" OAllowNull="True" ↙
        ↳ OInUpdate="True" OInWhere="True" OriginColName="↙
        ↳ Coords" SourceSize="18"/>
    <Column Name="Altitud" SourceName="Altitud" SourceID="4"↙
        ↳  DataType="Double" Searchable="True" AllowNull="↙
        ↳ True" Base="True" OAllowNull="True" OInUpdate="↙
        ↳ True" OInWhere="True" OriginColName="Altitud"/>
    <Column Name="Comentario" SourceName="Comentario" ↙
        ↳ SourceID="5" DataType="AnsiString" Size="50" ↙
        ↳ Searchable="True" AllowNull="True" Base="True" ↙
        ↳ OAllowNull="True" OInUpdate="True" OInWhere="True"↙
        ↳  OriginColName="Comentario" SourceSize="50"/>
  </ColumnList>
  <ConstraintList/>
  <ViewList/>
```

```
15    <RowList/>
16   </Table>
17   </TableList>
18   <RelationList/>
19   <UpdatesJournal>
20    <Changes/>
21   </UpdatesJournal>
22   </Manager>
23  </FDBS>
```

Listado 7.1 Archivo XML con el contenido del `TFDMemTable`

Podemos, por tanto, definir la estructura de las tablas de nuestra base de datos en memoria durante la fase de diseño, usando tantos `TFDMemTable` como necesitemos y estableciendo las relaciones que correspondan entre ellos, así como conectar esos componentes con la interfaz de usuario. De esta forma tendríamos un programa preparado para almacenar información sin estar conectado a una base de datos ni utilizar más archivos que el propio ejecutable.

> **NOTA**
>
> *Para que el contenido de un `TFDMemTable` creado según el procedimiento anterior estuviese disponible a través del motor de consultas SQL local, el componente `TFDLocalSQL`, tendremos que vincular al primero con el segundo. Para ello podemos usar la propiedad `LocalSQL` del `TFDMemTable` o bien la propiedad `DataSets` del `TFDLocalSQL`.*

7.3.2 Durante la ejecución

Si lo preferimos, podemos crear la estructura de nuestra base de datos en memoria durante la propia ejecución del programa, en lugar de hacerlo en la fase de diseño. En ese caso contamos también con varias alternativas, entre ellas: usar las propiedades y métodos del componente `TFDMemTable`, `TFDTable` o `TFDQuery`, o bien trabajar directamente sobre el motor local de consultas SQL usando sentencias DDL.

La colección `FieldDefs`

Los citados componentes cuentan con una propiedad, llamada `FieldDefs`, que contiene una colección de objetos con la definición de cada una de las columnas del conjunto de datos. Es posible agregar elementos a dicha colección mediante el método `Add`, al que debemos facilitar dos parámetros: el nombre de la columna y su tipo. Opcionalmente también puede facilitarse un tamaño y un valor booleano indicando si el campo ha de contener obligatoriamente un valor. De esta manera es posible definir las columnas que se precisen, tras lo cual usaríamos el método `CreateDataSet` para, efectivamente, crear el conjunto de datos usando las definiciones introducidas en `FieldDefs`.

Suponiendo que tuviésemos en nuestro módulo de datos o formulario un componente llamado `FDTable1`, con el código mostrado en el Listado 7.2 definiríamos una tabla con cinco columnas, la crearíamos y, finalmente, la enlazaríamos con el `TFDLocalSQL` para poder usarla mediante el motor local de SQL.

```
1  with FDTable1 do
2  begin
3    FieldDefs.Add('Lugar', ftInteger);
4    FieldDefs.Add('TimeStamp', ftDateTime);
5    FieldDefs.Add('Coords', ftString, 18);
6    FieldDefs.Add('Altitud', ftFloat);
7    FieldDefs.Add('Comentario', ftString, 40);
8
9    CreateDataSet;
10
11   FDLocalSQL1.DataSets.Add(FDTable1);
12 end;
```

Listado 7.2 Creación de una tabla en ejecución

En caso de que tuviésemos la información sobre la estructura de la tabla en un archivo XML, como el antes mostrado en el Listado 7.1, podríamos usar el método `LoadFromFile` del componente para cargar la estructura y, si los hubiese, también los datos iniciales. En este caso el procedimiento de creación implementado en el Listado 7.2 quedaría reducido a una única sentencia del tipo `FDTable1.LoadFromFile('Archivo.xml')`.

> **NOTA**
>
> *Además de definir la estructura del conjunto de datos, también podríamos introducir información inicial en el mismo. Para ello, tras la llamada a* `CreateDataSet`*, usaríamos el método* `AppendRecord` *facilitando el conjunto de valores a agregar.*

Creación de estructuras mediante sentencias DDL

El método alternativo para definir la estructura de una base de datos en memoria durante la ejecución del programa, sin necesidad de agregar componentes como `TFDMemTable`, `TFDTable` o `TFDQuery`, consiste en utilizar sentencias SQL de definición de datos. Estas sentencias se enviarán directamente al `TFDConnection`, mediante el método `ExecSQL`, encargándose el motor local de procesamiento de consultas de su ejecución y, como resultado, la creación de la tabla, vista u otro objeto que se quiera definir.

Además de sentencias DDL para definir estructuras, también podemos usar el método `ExecSQL` para introducir cualquier contenido inicial que deban contener las tablas. Por ejemplo:

```
1  FDConnection1.ExecSQL(
2    'CREATE TABLE Lugares (Codigo INT PRIMARY KEY, Nombre ↙
       ↳ TEXT)');
3
4  FDConnection1.ExecSQL(
5    'INSERT INTO Lugares VALUES (1, "Jaén")');
```

Listado 7.3 Creación de una tabla en memoria e inserción de datos

Siempre que el `TFDConnection` estuviese conectado a un componente `TFDLocalSQL`, el contenido generado de esta forma será accesible a través de componentes `TFDTable` y `TFDQuery`. No tenemos más que enlazarlos al mismo `TFDConnection` y seleccionar la tabla o introducir la consulta haciendo referencia a la tabla. Esos componentes, a su vez, serían los que enlazaríamos con la interfaz de usuario de la aplicación.

Obtención de la estructura desde otro conjunto de datos

La tercera opción que tenemos a nuestro alcance, a la hora de definir la estructura de una base de datos en memoria, consiste en copiar la estructura de conjuntos de datos ya existentes. Esto implica que deberíamos contar con algún componente conectado a una base de datos, desde la que obtendría su contenido. Suponiendo que fuese un `TFDQuery` con el resultado de una consulta, podríamos transferir todo su contenido, tanto estructura como datos, a un `TFDMemTable`, usando el método `CopyDataSet`. A continuación se muestra un ejemplo de cómo lo haríamos:

```
1 FDMemTable1.CopyDataSet(FDQuery1, [coStructure, coRestart, ↙
    ↳ coAppend]);
```

Listado 7.4 Titulo

El segundo parámetro de este método determina qué se copia: `coStructure` hace referencia a la estructura y `coAppend` a que los datos se agregarán también al objeto `FDMemTable1`. La opción `coRestart` indica al método que antes de iniciar la copia debe colocar el cursor de lectura al inicio del `FDQuery1`, obteniendo así todo su contenido.

7.4 Caso práctico

Ahora que ya conocemos las distintas vías que podemos seguir para crear una base de datos en memoria y trabajar con ella, pongamos ese conocimiento en práctica desarrollando un nuevo proyecto. Este sería el de una aplicación que hipotéticamente irá registrando de forma automática y periódica las coordenadas en que nos encontramos, para lo cual recurriría al sensor integrado en la mayoría de teléfonos móviles y tabletas. Además de mostrar esa información a medida que la registra, el programa debe permitir al usuario introducir un comentario, así como manipular las entradas generadas y ejecutar consultas sobre ellas. Todos los datos estarán alojados exclusivamente en memoria.

Creamos un nuevo proyecto vacío de tipo **MULTI-DEVICE APPLICATION**, lo cual nos permitiría desplegarlo en dispositivos móviles, y le agregamos un

módulo de datos. Enlazamos ambos elementos, como es habitual, para que desde el formulario podamos acceder a los componentes de datos. A partir de aquí iremos dando los pasos descritos en los siguientes apartados.

EJEMPLO 7.1 `BDDMemoria.dproj`

Puedes encontrar este proyecto completo en la carpeta `BDDMemoria`. No se precisa configuración adicional para poder utilizarlo ya que los datos residen en memoria.

7.4.1 Configuración de los componentes de datos

Introducimos en el módulo de datos del proyecto los componentes indicados a continuación:

- `TFDConnection`: Tras añadirlo usamos el Inspector de objetos para modificar sus propiedades `DriverName`, seleccionando `SQLite` de la lista desplegable, y `LoginPrompt`, a la que daremos el valor `False`.
- `TFDLocalSQL`: No tenemos más que insertarlo en el formulario y su propiedad `Connection` debería vincularlo automáticamente con el componente anterior.
- `TFDTable`: Al igual que el anterior, se enlazará automáticamente con el `TFDConnection`. Lo utilizaremos para definir la estructura de una tabla en memoria. Únicamente modificaremos su propiedad `TableName`, a la que asignaremos el valor `Rastro`. Este será el nombre que tenga la tabla en nuestra base de datos en memoria.
- `TFDQuery`: Su propiedad `Connection` también le enlazará con el componente `TFDConnection`. Nos servirá para ejecutar consultas arbitrarias sobre la base de datos en memoria. No tenemos que modificar ninguna de sus propiedades.

- `TFDGUIxWaitCursor`: Como ya sabemos, este componente es necesario en cualquier aplicación que utilice FireDAC.

Pretendemos crear la estructura de la base de datos en memoria, así como añadir alguna información inicial, al inicio de la ejecución del programa. Para ello introduciremos el código mostrado en el Listado 7.5 en el evento `OnCreate` del módulo de datos. La primera tabla, que almacenará el rastro de la ruta que estamos siguiendo, la creamos sirviéndonos del componente `TFDTable` a cuya propiedad `TableName` asignamos el valor `Rastro`. También agregamos un par de entradas en la tabla. Es importante no olvidar enlazarla con el componente `TFDLocalSQL`, de lo contrario no formaría parte de la base de datos en memoria y las consultas no la encontrarían. La segunda tabla, que contiene una lista de lugares, la creamos mediante sentencias SQL y, de la misma forma, le añadimos varias filas. Por último creamos una vista, enviando la sentencia SQL al `TFDConnection` mediante el método `ExecSQL`.

```
1  procedure TdmMemoria.DataModuleCreate(Sender: TObject);
2  begin
3    with FDTable1 do
4    begin
5      FieldDefs.Add('Lugar', ftInteger);
6      FieldDefs.Add('TimeStamp', ftDateTime);
7      FieldDefs.Add('Coords', ftString, 18);
8      FieldDefs.Add('Altitud', ftFloat);
9      FieldDefs.Add('Comentario', ftString, 40);
10
11     CreateDataSet;
12
13     AppendRecord([1, Now,'(37.779594, -3.784906)', 574, '↙
           ↳ Salida']);
14     AppendRecord([1, IncMinute(Now, 120), '(37.769031, ↙
           ↳ -3.807063)', 923, 'Castillo']);
15
16     FDLocalSQL1.DataSets.Add(FDTable1);
17
18     Open;
19   end;
20
```

```
21    with FDConnection1 do
22    begin
23      ExecSQL(
24       'CREATE TABLE Lugares (Codigo INT PRIMARY KEY, Nombre ↙
             ↳ TEXT)');
25      ExecSQL(
26       'INSERT INTO Lugares VALUES (1, "Jaén")');
27      ExecSQL(
28       'INSERT INTO Lugares VALUES (2, "Los Villares")');
29      ExecSQL(
30       'INSERT INTO Lugares VALUES (3, "Torredelcampo")');
31    end;
32
33    FDConnection1.ExecSQL(
34      'CREATE VIEW LugaresRastro AS ' +
35      ' SELECT Nombre, Altitud, Comentario ' +
36      ' FROM Rastro R ' +
37      ' INNER JOIN Lugares L ' +
38      '   ON R.Lugar=L.Codigo');
39  end;
```

Listado 7.5 Código que generará la base de datos en memoria

Una vez que se ejecute este código la base de datos estará disponible para la aplicación. Podemos utilizarla como lo haríamos con cualquier otra.

7.4.2 Diseño de la interfaz de usuario

El formulario de esta aplicación estará dividido en dos partes bien diferenciadas. En la superior tendremos una cuadrícula que mostrará la tabla `Rastro`, con la información que teóricamente se va añadiendo del sensor de posición. También dispondremos un `TBindNavigator` para facilitar las operaciones de edición sobre el contenido de la cuadrícula. La parte inferior del formulario tendrá otra cuadrícula, así como un `TEdit` y un `TButton`. El objetivo es facilitar la ejecución de cualquier consulta usando el `TQuery` que habíamos añadido al módulo de datos, mostrando el resultado obtenido en la cuadrícula.

La división en dos partes la efectuaremos introduciendo en el formulario dos controles `TPanel`, uno ajustado a la parte superior y el otro ocupando el resto del espacio disponible. Dentro de cada `TPanel` colocaremos el resto de los controles citados, tal y como se aprecia en la Figura 7.4. En la ventana de estructura, visible en el margen izquierdo, podemos apreciar la relación entre contenedores y controles insertados en ellos.

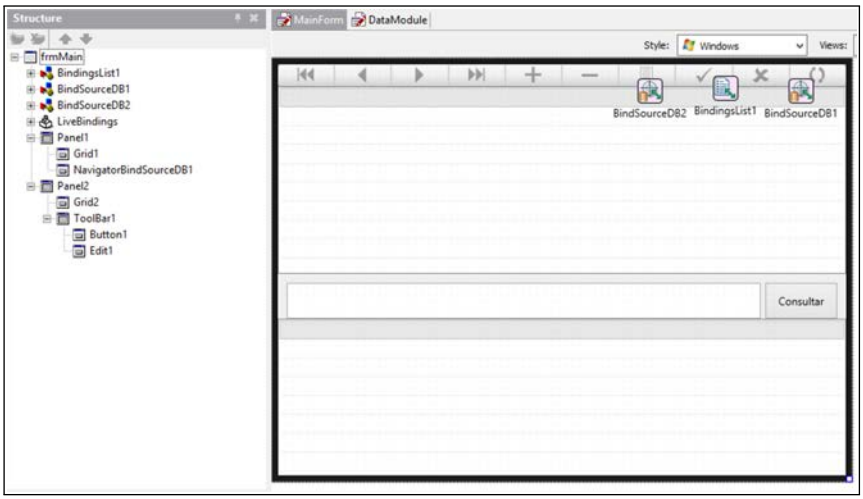


Figura 7.4 ESTRUCTURA DE LA INTERFAZ DE USUARIO

El paso siguiente será vincular las cuadrículas, los dos controles `TGrid`, con sus respectivos orígenes de datos. El superior lo conectaremos con el `TFDTable` y el inferior con el `TFDQuery`. Este trabajo podemos hacerlo agregando manualmente los `TBindSourceDB` y `TBindingsList` o bien usando el **LIVEBINDINGS DESIGNER** (véase la Figura 7.5) para establecer las conexiones mediante operaciones de arrastrar y soltar.

El enlace entre el `TGrid` superior y el `TBindNavigator` con el componente `TFDTable` facilitará tanto la visualización de los datos de la tabla `Rastro` como su edición. El `TGrid` inferior, por el contrario, no mostrará nada hasta en tanto no se ejecute alguna consulta sobre el `TFDQuery`. Para ello haremos doble clic sobre el `TButton` y añadiremos el código mostrado a continuación:

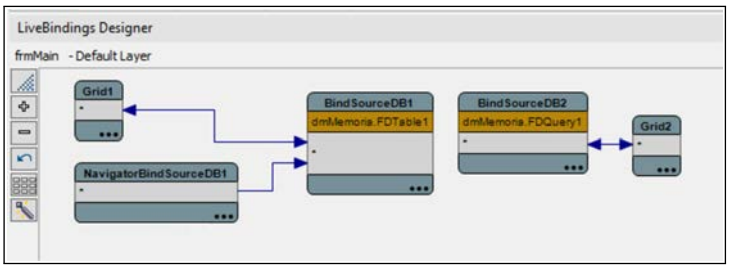


Figura 7.5 ENLACE DE LAS CUADRÍCULAS CON LOS COMPONENTES DE DATOS.

```
1  procedure TfrmMain.Button1Click(Sender: TObject);
2  begin
3    with dmMemoria.FDQuery1 do
4    begin
5      Close;
6      SQL.Clear;
7      SQL.Add(Edit1.Text);
8      Open;
9    end;
10 end;
```

Listado 7.6 Código asociado al botón de ejecución de consultas

Ahora ya estamos en disposición de ejecutar el programa y probar su funcionamiento (véase la Figura 7.6), editando el contenido de la cuadrícula superior y ejecutando consultas en la inferior.

En las consultas podemos hacer referencia tanto a las dos tablas que componen la base de datos en memoria como a la vista que las relaciona. Así, podríamos obtener una lista de los lugares registrados con la consulta `SELECT * FROM Lugares`, obtener una lista de todos los comentarios por orden cronológico con la consulta `SELECT Comentario FROM Rastro ORDER BY TimeStamp`, etc.

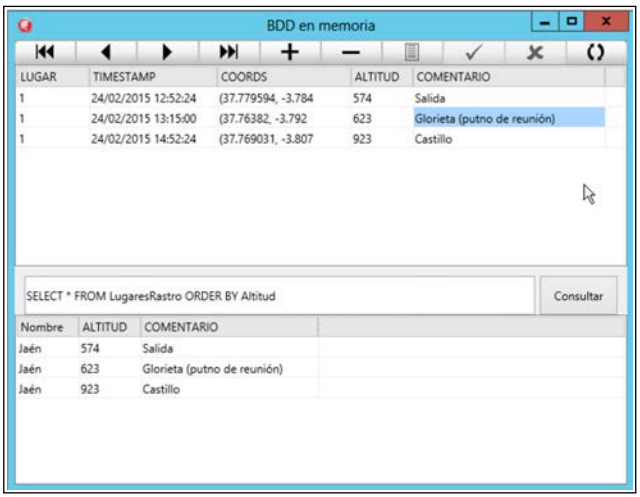


Figura 7.6 LA APLICACIÓN MOSTRANDO LA BASE DE DATOS EN MEMORIA

NOTA

Aunque no hemos incluido un `TBindNavigator` *asociado al* `TGrid` *inferior, esto no nos impediría modificar los datos que se están mostrando. Es perfectamente posible, por ejemplo, seleccionar el contenido de la tabla* `Lugares` *y modificar los códigos o nombres de los lugares.*

7.5 Exportación de los datos

El uso de una base de datos en memoria implica que cada vez que se salga de la aplicación la información se pierda, recreándose al volver a ejecutarse. Por ello es una opción adecuada para casos en los que, por regla general, los datos no serán necesarios nada más que durante la ejecución del programa.

Puede darse el caso, no obstante, en que al usuario le interese exportar los datos que ha obtenido para usarlos posteriormente, posiblemente desde otra aplicación. El supuesto registro efectuado por el ejercicio previo, por ejemplo, podría ser analizado después con otra aplicación para representar las coordenadas en un mapa o medir distancias.

Componentes como `TFDMemTable` y `TFDQuery` disponen de un método `SaveToFile`, con el que se crearía un archivo XML conteniendo la estructura y los datos. Es la función a la que se invoca con la opción **SAVE TO FILE** que se explicó anteriormente, al describir la definición de la estructura en la fase de diseño con un `TFDMemTable`. El inconveniente de este método es que únicamente guardará el contenido de un conjunto de datos, la información del `TFDTable` o `TFDQuery` en cuestión, pero no toda la base de datos.

7.5.1 El componente `TFDSQLiteBackup`

Dado que la base de datos en memoria se construye usando el controlador de SQLite, resulta posible utilizar componentes específicos de dicho controlador para operar sobre ella. Uno de esos componentes es `TFDSQLiteBackup`, cuya finalidad es realizar una copia de seguridad de una base de datos SQLite cualquiera.

Para utilizar este componente hemos de establecer la configuración de copia, llamando a continuación a su método `Backup`. La configuración se fija mediante las siguientes propiedades:

- `DriverLink`: Debemos asignar a esta propiedad una referencia al componente `TFDPhysSQLiteDriverLink` que esté utilizando la base de datos. Es un parámetro que podemos establecer desde el Inspector de objetos, abriendo la lista desplegable adjunta y seleccionando el componente que representa el controlador SQLite.
- `DatabaseObj` o `Database`: Origen de la copia de seguridad. La primera propiedad indica que se trata de un objeto, una base de datos en memoria, mientras que la segunda se usa cuando el origen es un archivo en disco.
- `DestDatabase` o `DestDatabaseObj`: Destino de la copia de seguridad. La primera propiedad se usa cuando el destino es un archivo en disco, mientras que la segunda usaría como destino un objeto en memoria.

El componente `TFDSQLiteBackup` generará el evento `OnProgress` mientras dure la operación de copia de datos desde el origen al destino, consultando las propiedades `PageCount` y `Remaining` para determinar el porcentaje de trabajo ya hecho.

7.5.2 En la práctica

Podemos agregar la posibilidad de exportar los datos usados por la anterior aplicación incluyendo en el módulo de datos un componente `TFDSQLiteBackup`, enlazándolo con el componente `TFDPhysSQLiteDriverLink` e implementando un método como el mostrado a continuación:

```
1  with FDSQLiteBackup1 do
2  begin
3    DatabaseObj := FDConnection1.CliObj;
4    DestDatabase := 'DatosRastro.sdb';
5    Backup;
6  end;
```

Listado 7.7 Exportación de la base de datos en memoria a un archivo

La propiedad `CliObj` de un `TFDConnection` devuelve el objeto interno asociado al controlador que está utilizando. En este caso facilitaría el objeto SQLite que gestiona la base de datos en memoria que, en definitiva, es lo que queremos salvaguardar.

Desde el formulario que actúa como interfaz de usuario llamaríamos a este método, por ejemplo al cerrar la aplicación o bien a demanda, agregando un nuevo botón a la interfaz. El archivo `DatosRastro.sdb` podríamos abrirlo después desde otra aplicación, como cualquier base de datos. De hecho, bastaría con definir una conexión desde el Explorador de datos de Delphi para poder abrir el archivo y explorar su contenido, accediendo incluso a los datos que contiene tal y como se muestra en la Figura 7.7.

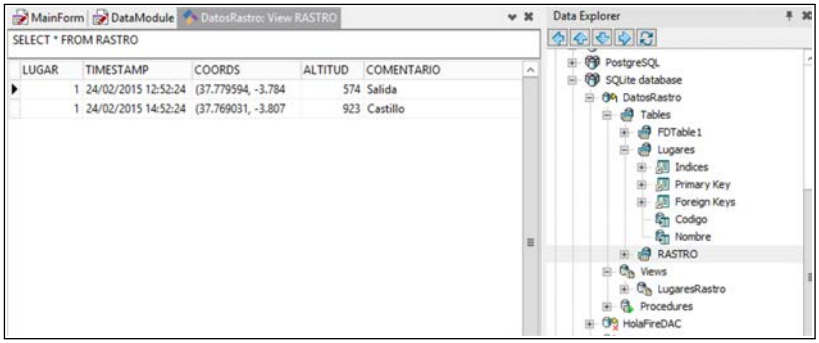


Figura 7.7 EXAMINAMOS LOS DATOS EXPORTADOS CON EL TFDSQLITEBACKUP

7.6 Resumen

Este capítulo nos ha servido para conocer una configuración adicional de trabajo con bases de datos mediante FireDAC. Hemos utilizado fundamentalmente componentes que ya conocíamos, pero operando únicamente con información alojada en memoria, sin una conexión real con una base de datos.

Además de para los supuestos mencionados anteriormente, esta configuración también nos serviría al trabajar con aplicaciones distribuidas, en las que el programa cliente obtiene datos de un servidor remoto, trabaja sobre ellos de manera desconectada, con una base de datos en memoria, y transfiere los cambios al final.

Una vez que hemos conocido las variantes más importantes de conexión a datos en aplicaciones mono-capa, en las que los datos residen en la misma máquina que ejecuta la aplicación, en el próximo capítulo, último de esta primera parte del libro, trataremos un tema totalmente distinto pero de vital importancia: el uso de Unicode con FireDAC.

Capítulo 8

BASES DE DATOS Y UNICODE

Las aplicaciones que desarrollemos, especialmente aquellas que cuentan con una interfaz web o están dirigidas a dispositivos móviles, es probable que sean utilizadas por personas que utilizan diferentes idiomas. Incluso si la interfaz de usuario únicamente está en un idioma concreto, por ejemplo en inglés, dichos usuarios necesitarán almacenar datos usando su propio idioma, lo cual puede plantearnos un cierto desafío si dicho idioma usa un alfabeto distinto al conocido habitualmente como *occidental*.

¿Está nuestro programa preparado para permitir el tratamiento de datos con caracteres de otros alfabetos? Si utilizamos una versión reciente de Delphi una

buena parte del trabajo ya lo tendremos hecho, ya que desde su versión 2009 los tipos de datos básicos para el almacenamiento de caracteres y cadenas de caracteres son Unicode. No obstante, es preciso determinar si la base de datos donde alojamos la información, así como los elementos de acceso a la misma, están correctamente configurados para facilitar el almacenamiento y recuperación de datos Unicode.

En este capítulo comenzaremos introduciendo brevemente qué es Unicode y sus diferentes codificaciones, familiarizándonos con la forma en que se codifican los caracteres y cadenas de caracteres en Delphi, para abordar después los detalles relativos al trabajo con datos Unicode en distintas bases de datos.

8.1 Introducción a Unicode

Los ordenadores almacenan toda la información con la que trabajamos como secuencias de números, incluyendo los textos. Esto hace necesario aplicar un proceso de codificación y descodificación cada vez que un carácter es almacenado o recuperado del almacenamiento. Dependiendo de qué asociación se establezca entre las secuencias numéricas y las grafías dependerá de que el resultado obtenido sea un texto que podamos leer o una nube ilegible de caracteres inconexos.

8.1.1 Estándares de codificación: ASCII

La codificación de caracteres mediante códigos numéricos ya se usaba antes de la llegada de los ordenadores. En estos se han utilizado distintas codificaciones, dependiendo de los países e incluso los fabricantes, hasta que emergieron estándares como EBCDIC (*Extended Binary Coded Decimal Interchange Code*) y ASCII (*American Standard Code for Information Interchange*). Este último fue el utilizado mayoritariamente en los ordenadores personales.

Originalmente el estándar ASCII usaba únicamente 7 bits, por lo que podían representarse un máximo de 128 caracteres distintos. De estos una parte, los primeros 32, estaban ocupados por *códigos de control*, mediante los que se indicaban acciones a los dispositivos tales como el avance de línea, retorno de carro, avance de página, etc. El resto eran suficientes para contener los números, letras

minúsculas y mayúsculas y algunos símbolos, por supuesto siempre que nuestro idioma fuese el inglés. Como puede apreciarse en la Figura 8.1, prácticamente no había espacio para más.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figura 8.1 CONJUNTO DE CARACTERES ASCII DE 7 BITS.

Dado que la práctica totalidad de ordenadores personales de los 70 e inicios de los 80 procesaban los datos en bloques de 8 bits, existía la posibilidad de extender el conjunto ASCII estándar agregando 128 caracteres adicionales. Así nació el conjunto de caracteres ASCII extendido. La tabla de grafías mostrada en la Figura 8.2 corresponde al conjunto extendido usado por el sistema operativo DOS en países de lengua hispana. No solo se incluían caracteres acentuados y la eñe, sino que había espacio adicional para otros símbolos y también algunos caracteres gráficos.

En realidad la parte extendida de ASCII se configuraba según conviniese a cada país, usando para ello distintas páginas de codificación con las que se incluían las grafías apropiadas. A pesar de ello seguían existiendo alfabetos que no tenían cabida en este sistema de codificación por su limitada capacidad, existiendo otros, por ejemplo en Japón, que utilizaban 16 o 32 bits para poder representar todos los símbolos que necesitaban.

Durante el tiempo en que los sistemas informáticos operaron como islas, sin conexiones hacia el exterior, el uso de diferentes sistemas de codificación no suponía un mayor problema. El intercambio de datos a escala global, sobre todo tras emerger la Web como plataforma de acceso a la información y aplicaciones, provocó que se buscara una solución capaz de acoger todos los caracteres de los distintos idiomas de nuestro planeta, ofreciendo una codificación unificada que eliminase las barreras existentes a la hora de compartir los datos.

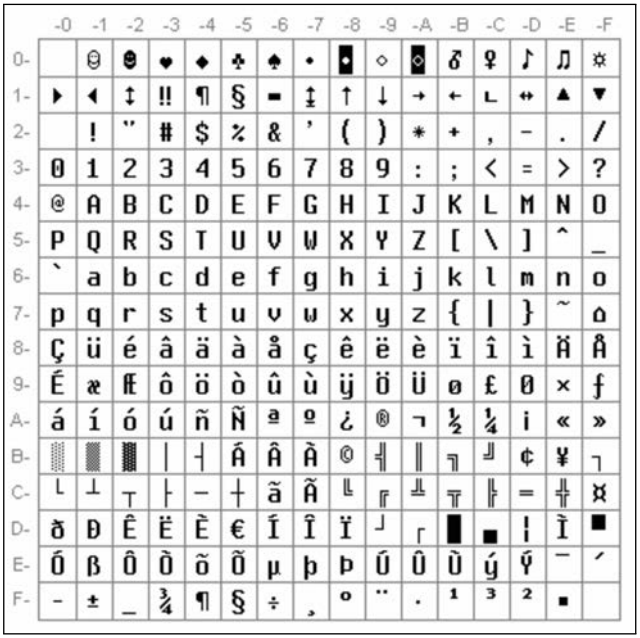


Figura 8.2 CONJUNTO DE CARACTERES ASCII EXTENDIDO DE 8 BITS

8.1.2 Unicode y las codificaciones UTF-N

El estándar Unicode nació con el objetivo de definir una codificación común para todos los caracteres y símbolos utilizados en cualquier lengua, eliminando así las barreras que suponían los distintos sistemas de codificación, incompatibles entre sí, utilizados históricamente por cada país, sistema hardware o software. Este ambicioso objetivo ha ido cumpliéndose paulatinamente, a lo largo de las diferentes versiones del estándar en las que se incluyen nuevos *code point*, denominación que reciben los códigos numéricos correspondientes a los caracteres y otra información asociada a los mismos.

La última versión de Unicode es la 7.0 y se compone de 112 956 elementos distintos, incluyendo caracteres de múltiples alfabetos[1] (véase la Figura 8.3), ideogramas, símbolos matemáticos, etc. Como es fácil deducir a partir de dicho número, se precisan 32 bits (4 bytes) para cada *code point*[2]. No obstante, los alfabetos de uso más común tienen asignados códigos numéricos representables con 16 bits o incluso con 8 bits. De hecho, el alfabeto estándar usado en inglés sigue coincidiendo con la codificación que tenía en ASCII, por lo que bastan 8 bits para cada carácter. Utilizar 4 bytes por carácter en estos casos representaría una ocupación innecesaria de memoria.

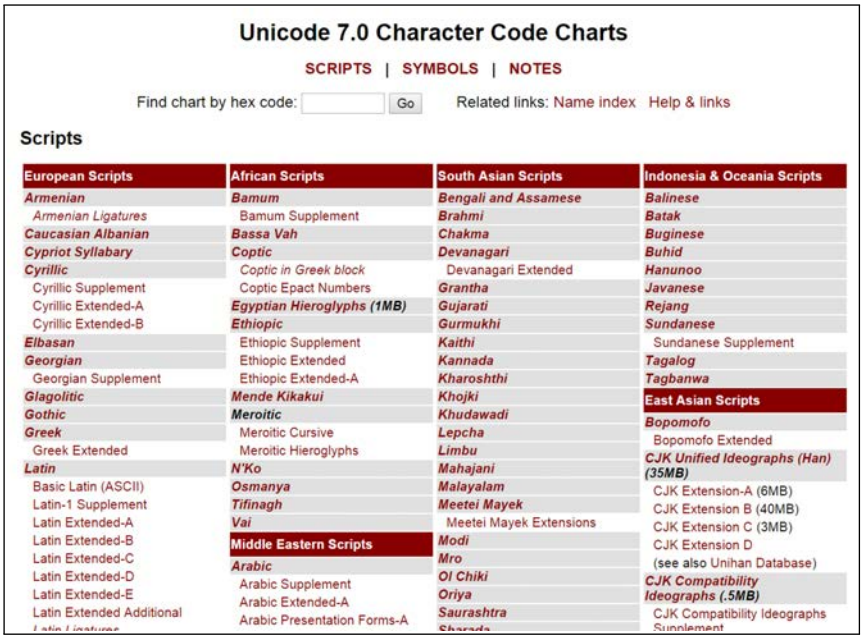


Figura 8.3 CONJUNTOS DE CARACTERES EXISTENTES EN UNICODE

[1]Podemos consultar los distintos conjuntos de caracteres existentes en Unicode, agrupados por zonas geográficas y lenguas, en `http://www.unicode.org/charts/`.

[2]La primera versión de Unicode utilizaba solo 16 bits, pero ante la imposibilidad de representar todos los alfabetos con esa capacidad se amplió a 21 bits en la versión 2.0.

NOTA

Unicode también asigna code points a elementos como los caracteres de control de ASCII y otro tipo de elementos que no son caracteres en sí, como las composiciones de varios símbolos, información de formato, etc.

Unicode estructura todos los elementos que es capaz de representar en subconjuntos de 65 536 unidades a los que denomina *planos*. En total existen 17 planos, por lo que teóricamente podrían asignarse códigos a más de un millón de caracteres y símbolos, si bien hay rangos que están reservados para el uso por parte de empresas y el propio consorcio que gestiona Unicode. Al primero de esos planos se le conoce habitualmente como BMP (*Basic Multilingual Plane*), siendo necesarios únicamente dos bytes para hacer referencia a cualquiera de sus elementos.

Con el objeto de optimizar la ocupación en memoria y, sobre todo, el tamaño de los documentos que almacenan información Unicode, existen tres formatos de transformación distintos (UTF, *Unicode Transformation Format*), conocidos como UTF-32, UTF-16 y UTF-8. Como es fácil deducir, el número final indica la cantidad de bits usada para representar cada elemento, si bien en los dos últimos casos esa cantidad es en realidad variable.

Cuando se utiliza UTF-32 toda la información sobre cada *code point* está alojada en los 4 bytes, por lo que no es precisa ningún procesamiento adicional. Una cadena de caracteres de este tipo ocuparía cuatro veces su longitud en bytes, lo cual representa en la mayoría de los casos, salvo que estemos almacenando información de idiomas asiáticos, un gasto innecesario de espacio en las bases de datos.

Mediante UTF-16 cada elemento ocupa 2 bytes salvo que haya que representar caracteres no occidentales, en cuyo caso se utiliza una pareja de elementos de 2 bytes (conocida como *surrogate pair*) para representarlo. En general, para las lenguas europeas el uso de UTF-16 supone utilizar la mitad de espacio que con UTF-32, sin ninguna limitación en cuanto a los caracteres y símbolos que es posible utilizar. La codificación y descodificación usando UTF-16 conlleva, a diferencia de UTF-32, seguir un cierto algoritmo, relativamente sencillo, en lugar de interpretar directamente el contenido.

El formato de transformación de UTF-8, que es el más usado en la Web y en general para el intercambio de datos, emplea un único byte para representar los primeros 256 caracteres. Estos coinciden con el mapa ISO-8859-1 (también conocido como *Latin1*) en el que aparecen la mayoría de caracteres europeos. Para caracteres fuera de este conjunto se utilizarán 2, 3 o 4 bytes, dependiendo del alfabeto al que pertenezcan. Por tanto UTF-8 ofrece la representación más compacta, por una parte, y también compatibilidad con ASCII y la codificación ISO-8859-1 que es probablemente la más utilizada. El mayor inconveniente de UTF-8 es que precisa seguir un procedimiento de codificación y descodificación algo más complejo que en los casos anteriores, por lo que su procesamiento requiere algo más de tiempo.

8.1.3 Unicode y cabecera BOM

Dado que Unicode codifica cada carácter/símbolo utilizando una secuencia de bytes, el orden de este es importante. Históricamente los procesadores han utilizado *Big Endian* o *Little Endian* dependiendo de su arquitectura (RISC o CISC) y del fabricante. Al intercambiar documentos que usan UTF-32 o UTF-16 es importante saber cuál es ese orden, ya que de lo contrario el receptor de la información podría obtener un documento indescifrable en lugar de uno legible. Con UTF-8 no existe este problema por la forma en que está definido el formato de transformación.

Los archivos codificados con UTF-32 cuentan con un primer elemento (4 bytes), denominado BOM (*Byte Order Mark*), con el que se indica si el orden de los bytes es *Big Endian* (`00 00 FE FF`) o *Little Endian* (`FF FE 00 00`). En el caso de UTF-16 el BOM ocupa 2 bytes y será `FE FF` o `FF FE`, respectivamente.

A pesar de que UTF-8 no precisaría de un BOM al inicio del archivo, hay editores que lo incluyen únicamente como firma UTF-8. El **BLOC DE NOTAS** de Windows, por ejemplo, agrega automáticamente el BOM si se elige como codificación Unicode o UTF-8 al guardar un archivo (véase la Figura 8.4). El BOM en este caso ocupa 3 bytes: `EF BB BF`. La mayoría de los editores de texto, incluyendo el de Delphi, interpretan adecuadamente el BOM y abren el archivo como UTF-8, por lo que nunca veremos esa firma sino el contenido correcto del archivo. No obstante, podemos utilizar un editor binario, como el que aparece en la Figura 8.5, para examinar el contenido real del archivo,

sin ningún tipo de tratamiento o interpretación. Ahí sí podremos verificar la existencia del BOM.

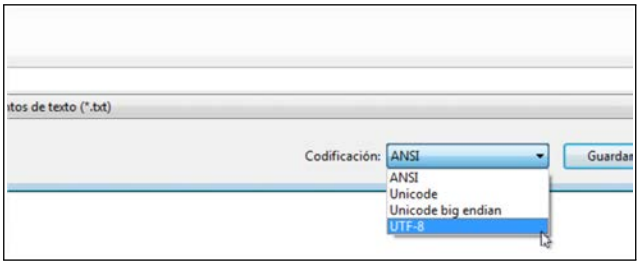


Figura 8.4 OPCIONES DE CODIFICACIÓN DEL BLOC DE NOTAS DE WINDOWS

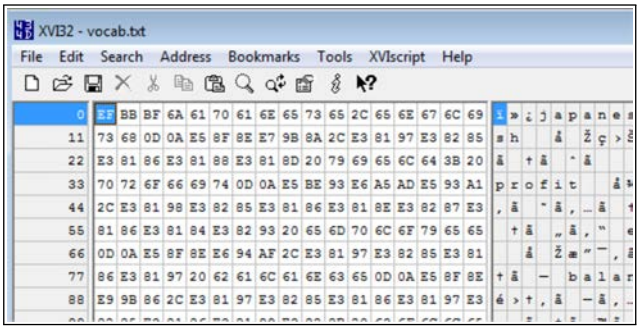


Figura 8.5 COMPROBAMOS EL BOM AÑADIDO POR EL BLOC DE NOTAS

8.2 Soporte Unicode en Delphi

Delphi ha contado con la capacidad de operar con cadenas de caracteres en las que cada elemento se representa con 16 bits desde la versión 3, en la que se introdujo el tipo `WideString`. Este facilitaba el trabajo con servicios de Windows como OLE (*Object Linking and Embedding*) y el modelo de objetos COM (*Component Object Model*) de Microsoft. Asociados al mismo existen los tipos

de datos `WideChar`, para almacenar un carácter con 16 bits de almacenamiento, y `PWideChar`, como puntero a una secuencia de caracteres `WideChar`. Actualmente el tipo `WideString` se mantiene en Delphi por compatibilidad hacia atrás, pero raramente nos encontraremos en la necesidad de utilizarlo. El tipo de cadena de uso preferente, y por defecto, es `UnicodeString`.

> **NOTA**
>
> *Los compiladores Delphi para iOS y Android no soportan el tipo de dato `WideString`, pero si el resto de tipos de cadenas de caracteres.*

Los componentes de VCL y FMX están preparados para trabajar con Unicode, ya que todas las propiedades y parámetros de métodos y eventos usan el citado tipo `UnicodeString`. Esto incluye a los componentes de conexión a bases de datos, así como a los servicios de lectura y escritura de información en archivos de la RTL.

Dado que el compilador y el entorno de Delphi están programados principalmente en Delphi, el soporte de Unicode se extiende a todos los elementos y podemos abrir y guardar archivos de texto Unicode, algo básico, pero también utilizar cualquier carácter Unicode como parte de identificadores de variables (véase la Figura 8.6), métodos, etc. Esto permite escribir el código usando los símbolos de nuestro propio idioma, no estando limitados a los usados en inglés como ocurría en el pasado.

Figura 8.6 UN IDENTIFICADOR DE VARIABLE EN JAPONÉS

8.2.1 Tipos de cadenas y caracteres

Desde la versión 2009 Delphi cuenta con un nuevo tipo de dato para trabajar con cadenas de caracteres, `UnicodeString`, que se une a las que ya existían de antes: `AnsiString` y `WideString`. La diferencia fundamental entre `UnicodeString` y `WideString` es que su funcionamiento interno, a pesar de almacenar el texto como UTF-16, se asemeja más al de `AnsiString` y al de las cadenas en Pascal en general. Esto significa que tiene un contador de referencias interno y puede ser usado como siempre hemos hecho con el `String` básico de Pascal.

El tipo `String` es en Delphi un sinónimo de `UnicodeString`. En consecuencia, cualquier aplicación en la que utilizásemos el tipo `String` al compilarla con una versión reciente de Delphi pasará a trabajar automáticamente con cadenas Unicode. El resto de los tipos se mantienen por compatibilidad. Si realmente necesitamos trabajar con cadenas de caracteres ASCII, no tenemos más que usar el tipo `AnsiString` en lugar de `String`.

Análogamente, los tipos `Char` y `PChar` son ahora sinónimos de `WideChar` y `PWideChar`. Si necesitamos trabajar con caracteres exclusivamente ASCII debemos usar el tipo `AnsiChar` y, para cadenas ASCII terminadas con nulo, el tipo `PAnsiString`.

Podemos apreciar la diferencia entre las cadenas `UnicodeString` (o simplemente `String`) y las `AnsiString` con un sencillo ejercicio. Introduce en un formulario Delphi tres `TLabel` y tres `TEdit` para introducir una cadena Unicode y mostrar su longitud y ocupación y los mismos elementos, por ejemplo a la derecha, para hacer lo mismo con cadenas ANSI. Para obtener la longitud de una cadena en caracteres seguiremos usando el método `Length`. La ocupación en bytes la calcularemos multiplicando esa longitud por el valor devuelto por la función `StringElementSize`. Esta toma como parámetro una cadena y devuelve la ocupación en bytes de cada uno de sus elementos.

El texto que se introduzca en los componentes `TEdit` siempre será Unicode, ya que su propiedad `Text` es de tipo `String`. Para poder trabajar con cadenas no Unicode necesitamos utilizar una variable de tipo `AnsiString`. Por ello vamos a añadir en la parte privada de la clase la declaración de dos variables, una de tipo `AnsiString` y otra de tipo `String`, que usaremos como almacenamiento de los datos introducidos en los controles `TEdit`:

```
...
 private
   { Private declarations }
   cadAnsi: AnsiString;
   cadUnicode: String;
...
```

Listado 8.1 Declaración de variables

EJEMPLO 8.1 `TiposUnicode.dproj`

Puedes encontrar este proyecto completo en la carpeta `TiposUnicode`. No se precisa ninguna configuración adicional para poder compilar y ejecutar.

Para actualizar la información mostrada en el formulario aprovecharemos los eventos `OnChange` de los `TEdit` en los que se introducirá el texto, básicamente trasladando el contenido de la propiedad `Text` a la variable correspondiente de las antes declaradas. Por último se invocará a un método interno que se encargará de actualizar los datos sobre longitud y ocupación. El código de estas operaciones es el mostrado a continuación:

```
procedure TForm1.UpdateGUI;
begin
  edUnicodeString.Text := cadUnicode;
  edAnsiString.Text := cadAnsi;

  longUnicode.Text := IntToStr(length(cadUnicode));
  bytesUnicode.Text := IntToStr(StringElementSize(cadUnicode↙
      ↳ ) * Length(cadUnicode));
  longAnsi.Text := IntToStr(length(cadAnsi));
  bytesAnsi.Text := IntToStr(Length(cadAnsi) * ↙
      ↳ StringElementSize(cadAnsi));
end;
```

```
11
12 procedure TForm1.edAnsiStringChange(Sender: TObject);
13 begin
14   cadAnsi := edAnsiString.Text;
15   UpdateGUI;
16 end;
17
18 procedure TForm1.edUnicodeStringChange(Sender: TObject);
19 begin
20   cadUnicode := edUnicodeString.Text;
21   UpdateGUI;
22 end;
```

Listado 8.2 Actualización de datos de longitud y ocupación de las cadenas

La reasignación del contenido de las variables a la propiedad `Text` de los `TEdit` tiene por finalidad reflejar la información con la que realmente se está trabajando. El segundo `TEdit` permitiría la introducción de caracteres Unicode, pero al asignar su propiedad `Text` a la variable de tipo `AnsiString` parte de la información se perdería, por ello volvemos a asignar la variable a la citada propiedad. Es algo que podemos comprobar escribiendo algún carácter no occidental en dicho control, comprobaremos cómo se pierde y en su lugar aparece el carácter `?`. En la Figura 8.7 puede verse el programa en funcionamiento.

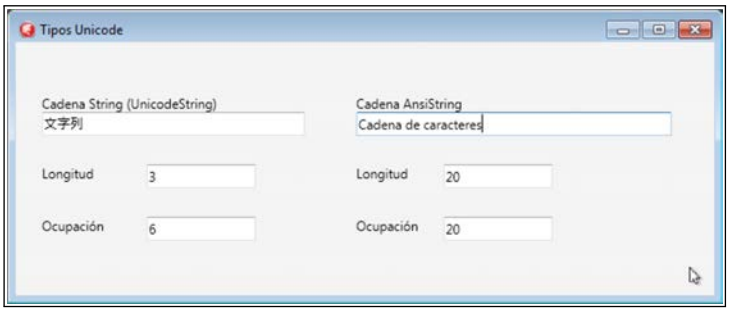


Figura 8.7 DIFERENCIAS ENTRE UNA CADENA UNICODE Y UNA ANSI

NOTA

La ocupación real de una de cadena de caracteres en Delphi es algo superior a la que aparece en este ejercicio, ya que en él no se ha considerado la cabecera que contiene datos como la longitud de la cadena, el contador de referencias a la misma, el tamaño de cada elemento, etc. Puedes encontrar información sobre la estructura de dicha cabecera en `http://docwiki.embarcadero.com/RADStudio/XE8/en/Unicode_in_RAD_Studio`.

Al igual que hemos hecho siempre, podemos acceder a un carácter individual de una cadena utilizando la sintaxis `varCadena[índice]`. Hemos de tener en cuenta, no obstante, que cada elemento no ocupa un byte, por lo que el índice no actúa como un desplazamiento en bytes desde el inicio de la cadena. Cualquier código que tengamos en el que se asuma que cada elemento ocupa un byte deberíamos revisarlo y corregirlo ya que probablemente no genere el resultado que debería. Asimismo no podemos asumir que el valor devuelto por `Length` es la ocupación de los caracteres de la cadena en bytes, por ejemplo a fin de reservar memoria o indicar el número de bytes a transferir. Todas estas son operaciones potencialmente peligrosas en las que dichas suposiciones han de eliminarse.

La conclusión, llegados a este punto, es que Delphi está preparado para trabajar con datos Unicode siempre que usemos los tipos por defecto, `Char` y `String`, que son los mismos que se usan en propiedades y parámetros de los componentes VCL/FMX. No hay ningún problema, por tanto, en crear una interfaz de usuario que acepte entradas en cualquier idioma y poder almacenar esos datos internamente. Los obstáculos pueden surgir a la hora de almacenar esa información o de recuperarla de un fuente externa, casos en los que podría ser necesario aplicar la conversión adecuada.

8.2.2 Conversión entre codificaciones

Las cadenas ANSI/ASCII pueden usar distintas páginas de códigos, utilizándose por defecto aquella establecida por la configuración local del sistema. A pesar de que el número de caracteres usados a la vez es limitado, seleccionando la página

adecuada es posible representar prácticamente cualquier carácter o símbolo. A partir del tipo `AnsiString` podemos definir tipos de cadena ANSI que utilicen páginas concretas sin más que especificar su código, por ejemplo:

```
type
  // Página de códigos ISO-2022-JP para Japonés
  JapString = type AnsiString(50220);

...

procedure TForm1.FormCreate(Sender: TObject);
var
  ansiJap: JapString;
  uniJap: String;
begin
  uniJap := 'Caracteres en japonés';
  ansiJap := uniJap;

  ShowMessage('uniJap: ' + uniJap + #13#10 + 'ansiJap: ' + ansiJap);
end;
```

Listado 8.3 Titulo

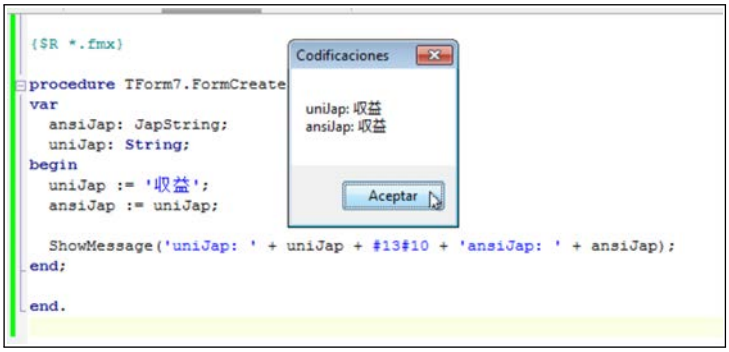


Figura 8.8 USO DE CADENAS ANSI CON PÁGINAS DE CÓDIGOS ALTERNATIVAS

Este programa se supone que tiene unos datos en japonés almacenados en una variable `String`, de tipo Unicode, pero que precisa convertir a ANSI por alguna razón. Como podemos apreciar, una simple asignación ha sido suficiente para conseguir los mismos datos en un `AnsiString` con la configuración adecuada.

> **EJEMPLO 8.2** `Codificaciones.dproj`
>
> Puedes encontrar este proyecto completo en la carpeta `Codificaciones`. No se precisa ninguna configuración adicional para poder compilar y ejecutar.

Puedes encontrar una lista de códigos de página en `http://www.iana.org/assignments/character-sets/character-sets.xhtml`. El inconveniente de este tipo de representación, frente a Unicode, es que resulta indispensable conocer el código de página en que se encuentra un cierto texto para poder interpretarlo adecuadamente. No obstante, puede ocurrir que una aplicación tenga que tratar con datos no Unicode que corresponden a una página concreta de códigos, siendo la anterior una solución para tratar con dicha información adecuadamente.

La clase `TEncoding`

La RTL nos ofrece, en el módulo `System.Utils`, una clase cuya finalidad es definir de manera explícita la codificación de un carácter o de una cadena: la clase `TEncoding`. Esta es una alternativa generalmente más adecuada para convertir datos entre distintas representaciones, ya sea en memoria, al recuperarlos desde archivos o al almacenarlos.

Mediante el método `GetEncoding` de la clase `TEncoding` podemos crear un objeto asociado a una codificación concreta, facilitando como parámetro el código que tiene asociado. Existen objetos ya predefinidos asociados a las codificaciones más comunes. Podemos obtenerlos mediante propiedades de la clase `TEncoding` como `UTF8`, `Unicode` o `ANSI`. La propiedad `Default` devuelve un objeto `TEncoding` con codificación ANSI asociada a la página de códigos actual en el caso de Windows.

Una vez que tenemos un objeto `TEncoding`, con la codificación que nos interese, podemos utilizar sus métodos para recuperar la secuencia de bytes que corresponde a una cadena o, a la inversa, obtener una cadena a partir de una secuencia de bytes con la codificación adecuada. Esta es la finalidad de los métodos `GetBytes` y `GetString`, respectivamente. Con el método `GetByteCount` se determina la longitud de la secuencia de bytes generados a partir de una cadena.

Las secuencias de bytes, que no son más que vectores del tipo `Byte`, pueden ser escritas y leídas de archivos, enviadas y recibidas por un canal de transmisión de datos y, operando sobre ellas en memoria, interpretarse de una forma u otra según convenga. El método `GetBytes` toma como argumento una cadena de caracteres en la codificación para la que se ha configurado el objeto `TEncoding`, devolviendo como resultado un valor de tipo `TBytes`, cuya definición es `TArray<Byte>`. Análogamente, `GetString` toma como entrada la secuencia de bytes y construye la cadena en la codificación adecuada, devolviéndola como un valor de tipo `String`.

Usando el método `Convert` de `TEncoding` es posible convertir entre diferentes tipos de codificaciones. Precisa tres parámetros como entrada: un objeto `TEncoding` indicando la codificación en que se encuentra actualmente la información, otro objeto del mismo tipo especificando la codificación de salida y, por último, la secuencia de bytes de la cadena a convertir. El valor devuelto como resultado será la secuencia de bytes en la nueva codificación. Asumiendo que tenemos las variables `ansiJap` y `uniJap` del ejercicio previo, el ejemplo siguiente, que forma parte del mismo proyecto, muestra cómo realizar la conversión de una codificación a otra mediante el citado método `Convert`.

```
1  var
2    AnsiJapones, Unicode: TEncoding;
3
4  begin
5    AnsiJapones := TEncoding.GetEncoding(50220);
6    Unicode := TEncoding.Unicode;
7
8    ansiJap := AnsiJapones.GetString( TEncoding.Convert(
9       Unicode, AnsiJapones, Unicode.GetBytes(uniJap)));
10
11   ShowMessage(ansiJap + ' (ANSI) -> ' +
```

```
12                IntToStr(AnsiJapones.GetByteCount(ansiJap))
13                + '␣bytes' + #13#10 +
14                uniJap + '␣(Unicode)->␣' +
15                IntToStr(Unicode.GetByteCount(uniJap)) +
16                '␣bytes');
```

Listado 8.4 Conversión entre codificaciones

La última sentencia de este ejemplo utiliza el método `GetByteCount` para mostrar la ocupación total del texto en cada codificación. Como puede apreciarse en la Figura 8.9, contando todos los elementos que conforman la cadena la representación en ANSI finalmente ocupa más que la cadena codificada como UTF-16.

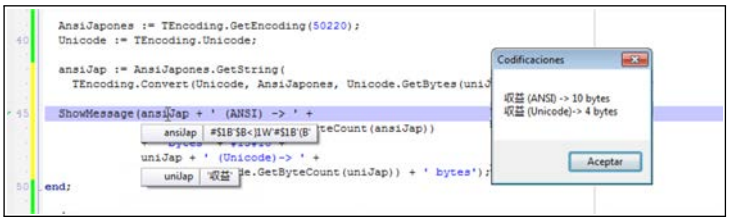


Figura 8.9 CONTENIDO REAL DE CADA VARIABLE

8.2.3 Lectura/Escritura de archivos Unicode

La clase `TEncoding` también nos resultará de utilidad a la hora de leer información de archivos y almacenarla. En el primer caso podemos detectar la codificación del texto, en caso de que no la conociésemos y el archivo contuviese BOM, mientras que en el segundo especificaríamos la codificación de almacenamiento. Con este fin recurriríamos a los siguientes métodos de `TEncoding`:

- `GetBufferEncoding`: Tomando como primer argumento una secuencia de bytes, correspondiente a una cadena, este método intenta detectar la codificación y la longitud del preámbulo (la cabecera con el BOM). El segundo parámetro será un objeto de tipo `TEncoding` y con valor `nil` o una codificación a comprobar. Si el valor es `nil` el método devolverá en ese

mismo parámetro la codificación detectada[3], en caso contrario solamente se verificará si el contenido coincide con la codificación indicada. El valor de retorno de la función facilita la longitud del preámbulo.

- `GetPreamble`: Devuelve como resultado una secuencia de bytes con el preámbulo correspondiente a la codificación asociada al objeto `TEncoding`. La finalidad es facilitar su escritura al inicio de un archivo.

NOTA

Si conocemos la codificación de un archivo que vamos a leer o escribir, podemos establecerla directamente al invocar a métodos como `LoadFromFile` y `SaveToFile`. Estos aceptan como segundo argumento un parámetro de tipo `TEncoding`.

Veamos en la práctica cómo leer un archivo de texto sin conocer la codificación, pero asumiendo que será UTF-16, UTF-8 o, en su defecto, ANSI con la página de códigos del sistema; cómo cambiar dicha codificación y cómo guardarlo nuevamente.

EJEMPLO 8.3 `CodArchivo.dproj`

Puedes encontrar este proyecto completo en la carpeta `CodArchivo`. No se precisa ninguna configuración adicional para poder compilar y ejecutar. Necesitaras al menos un archivo con codificación UTF-8 o UTF-16 para realizar pruebas.

Comenzaremos diseñando una interfaz de usuario como la mostrada en la Figura 8.10. En la parte superior hemos dispuesto un `TToolBar` con dos `TButton`, tres `TRadioButton` y un `TCheckBox`, usando los textos que se ven para la

[3]Este método es en realidad muy básico y solamente verifica si los primeros tres o cuatro bytes existentes en la secuencia coinciden con las firmas de UTF-8 o UTF-16. En el resto de los casos no sería de utilidad.

propiedad `Text` de cada uno de ellos. El resto del espacio está ocupado por un control `TMemo` que servirá para mostrar el texto. También se han incluido un componente `TOpenDialog` y un `TSaveDialog`. Los usaremos para facilitar la selección del archivo a abrir y el nombre del archivo al guardarlo, respectivamente.

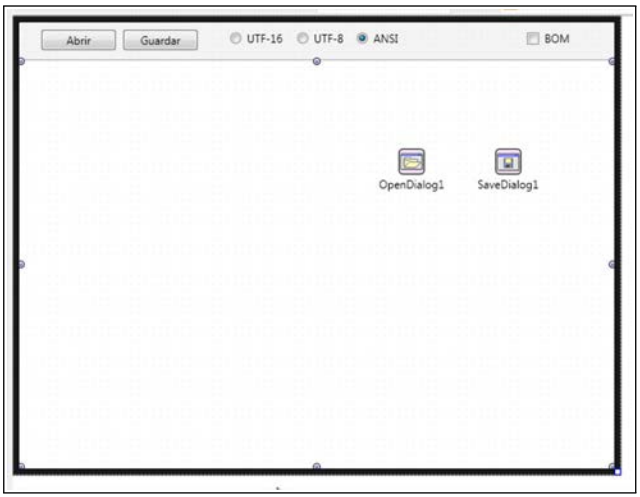


Figura 8.10 GUI DEL PROGRAMA PARA CARGAR/GUARDAR CON CODIFICACIÓN

Completaremos la definición de la clase del formulario, derivada de `TForm`, agregando los miembros necesarios para mantener en memoria el contenido del archivo que se haya cargado, así como la codificación que le corresponde y la longitud del preámbulo si es que cuenta con él. También se incluirán dos métodos adicionales que vamos a implementar de inmediato:

```
1  private
2    contenido: TBytes;
3    codActual: TEncoding;
4    longPreambulo: Integer;
5    function getEncoding: TEncoding;
6    procedure setEncoding(encoding: TEncoding);
```

Listado 8.5 Miembros a agregar a la clase derivada de `TForm`

La finalidad del método `getEncoding` es devolver un objeto `TEncoding` u otro dependiendo del `TRadioButton` que haya activo en la interfaz de usuario. Complementariamente, el método `setEncoding` recibe un objeto `TEncoding` como parámetro y activa el botón de radio correspondiente. La implementación de esos dos métodos es la mostrada a continuación:

```
function TfrmMain.getEncoding;
begin
  if UTF16.IsChecked then
    result := TEncoding.Unicode
  else if UTF8.IsChecked then
       result := TEncoding.UTF8
       else result := TEncoding.ANSI;
end;

procedure TfrmMain.setEncoding(encoding: TEncoding);
begin
  UTF16.IsChecked := encoding = TEncoding.Unicode;
  UTF8.IsChecked := encoding = TEncoding.UTF8;
  ANSI.IsChecked := encoding = TEncoding.ANSI;
end;
```

Listado 8.6 Métodos para obtener y establecer la codificación

Al hacer clic sobre el primero de los `TButton` el programa permitirá cargar cualquier archivo de texto, detectar su codificación y mostrar su contenido en el `TMemo`. Tras ajustar el tamaño del vector `contenido` a partir de la información obtenida del archivo y cargar en ella todo su contenido, recurriremos al método `GetBufferEncoding` para determinar la codificación actual. Si el preámbulo tiene una longitud superior a 0 bytes esto indicará que existe un BOM, por lo que activamos o desactivamos el `TCheckBox` en función de esta condición. Finalmente mostramos el contenido y activamos el botón de radio correspondiente para indicar la codificación:

```
procedure TfrmMain.btnAbrirClick(Sender: TObject);
var
  archivo: TFileStream;
```

```
4  begin
5    if OpenDialog1.Execute then
6    begin
7      archivo := TFileStream.Create(OpenDialog1.FileName, ↙
           ↳ fmOpenRead);
8      SetLength(contenido, archivo.Size);
9      archivo.ReadBuffer(Pointer(contenido)^, Length(contenido↙
           ↳ ));
10     archivo.Free;
11     codActual := nil;
12     longPreambulo := TEncoding.GetBufferEncoding(contenido, ↙
           ↳ codActual);
13
14     BOM.IsChecked := longPreambulo >  0;
15
16     Memo1.Text := codActual.GetString(contenido);
17     setEncoding(codActual);
18   end;
19 end;
```

Listado 8.7 Carga de un archivo y detección de la codificación

El evento `OnChange` de los tres controles `TRadioButton` estará conectado a un mismo evento. Su finalidad será convertir el texto de su codificación original a la actualmente seleccionada y mostrarlo de nuevo en el `TMemo`. Para ello usaremos el siguiente código:

```
1  procedure TfrmMain.ANSIChange(Sender: TObject);
2  var
3    codNueva: TEncoding;
4  begin
5    codNueva := getEncoding;
6
7    Memo1.Text := codNueva.GetString(
8        TEncoding.Convert(codActual, codNueva, contenido));
9  end;
```

Listado 8.8 Conversión entre codificaciones

Esta conversión puede no tener efecto alguno en lo que estamos viendo en el formulario si la conversión no implica pérdida de información. Es lo que ocurre al convertir de ANSI a UTF-8 o de UTF-8 a UTF-16. La conversión de un texto que originalmente usa alguna de estas dos últimas codificaciones a ANSI, por el contrario, sí que puede causar una pérdida de información y que el texto mostrado en el `TMemo` cambie. Volviendo a seleccionar la codificación original recuperaríamos el texto.

En realidad la finalidad principal de los `TRadioButton` y `TCheckBox` es indicar la configuración del archivo leído originalmente, por una parte, y establecer la que deseamos utilizar al guardar la información de nuevo en un archivo. Esto ocurrirá al pulsar el segundo `TButton`, ejecutando el siguiente código:

```
procedure TfrmMain.btnGuardarClick(Sender: TObject);
var
  archivo: TFileStream;
  codNueva: TEncoding;
  preambulo, contArchivo: TBytes;
  despContenido: Integer;
begin
  if SaveDialog1.Execute then
  begin
    codNueva := getEncoding;
    archivo := TFileStream.Create(SaveDialog1.FileName, ↙
        ↳ fmCreate);
    if BOM.IsChecked then
    begin
      preambulo := codNueva.GetPreamble;
      archivo.Write(preambulo[0], Length(preambulo));
    end;
    contArchivo := TEncoding.Convert(codActual, codNueva,
       contenido, longPreambulo,
       Length(contenido) - longPreambulo);
    archivo.Write(contArchivo[0], Length(contArchivo));
    archivo.Free;
  end;
end;
```

Listado 8.9 Guardar el texto con la configuración establecida

Tras crear el archivo, con el nombre que se haya indicado en el cuadro de diálogo abierto por el `TSaveDialog`, se escribe en él el preámbulo en caso de que el `TCheckBox` esté marcado. A continuación se convierte la información de su codificación original a la actualmente seleccionada, escribiendo el resultado en el archivo.

Para probar el programa (véase la Figura 8.11) lo mejor es partir de un archivo en codificación UTF-8 o UTF-16 con BOM, de forma que al abrirlo se detecte adecuadamente. A continuación podemos guardarlo en UTF-8 sin BOM o en ANSI y usar un editor hexadecimal para examinar el preámbulo o su ausencia, según los casos.

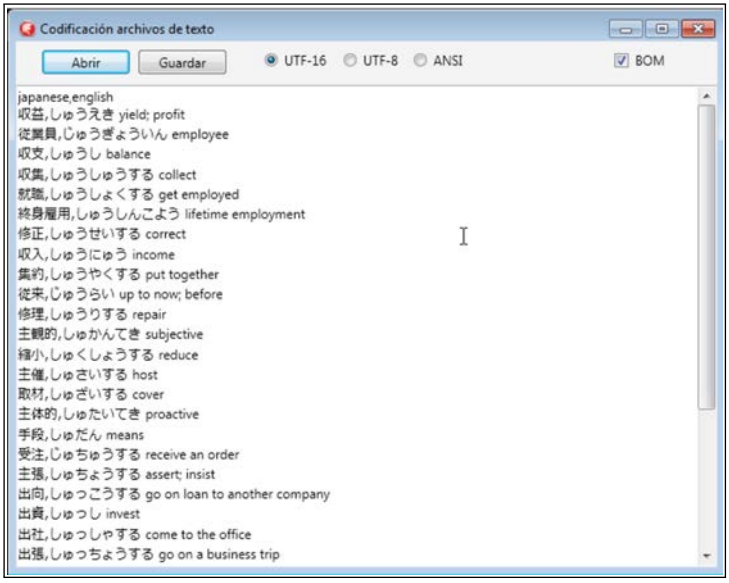


Figura 8.11 EL PROGRAMA EN FUNCIONAMIENTO MOSTRANDO LA CONFIGURACIÓN DE UN ARCHIVO RECIÉN ABIERTO

Si el archivo que interesa cargar usa una página de códigos ANSI distinta de la fijada por defecto por el sistema el texto no aparecerá legible en este programa, ya que no es capaz de determinar esa información por sí solo.

NOTA

En caso de que utilicemos los métodos `LoadFromFile` *y* `SaveToFile` *de un objeto* `TStrings`*, por ejemplo el asociado a la propiedad* `Lines` *del* `TMemo`*, además de poder indicar la codificación a usar también es posible establecer mediante la propiedad* `WriteBOM`*, de tipo* `Boolean`*, si ha de escribirse o no el BOM en los archivos.*

8.3 Bases de datos y Unicode

A diferencia de lo que ocurre cuando se opera con información procedente de archivos de distintos orígenes, cada uno de los cuales podría emplear una codificación distinta, al trabajar con una base de datos toda la información utilizará la misma codificación. Si es necesario efectuar conversiones, por ejemplo entre la codificación empleada por el RDBMS y la usada por un programa que actúa como interfaz de usuario de acceso a la misma, estas serán llevadas a cabo automáticamente por el controlador o los componentes FireDAC, según los casos. Para ello, no obstante, hemos de establecer la configuración adecuada en cada uno de los elementos que participan en una solución de este tipo. Esa será nuestra responsabilidad.

8.3.1 Configuración Unicode en FireDAC

Los componentes FireDAC de las actuales versiones de Delphi, entendiendo aquellas posteriores a 2009, están preparadas para realizar automáticamente la conversión entre codificaciones de caracteres cuando sea preciso. Internamente las consultas y parámetros a enviar al gestor de bases de datos, en propiedades como `SQL` (componente `TFDQuery`) y `CommandText` (`TFDCommand`), se almacenan siempre como Unicode. Esto nos permite, por ejemplo, introducir en una consulta texto en idiomas que usan otros alfabetos distintos al nuestro. Dependiendo de la configuración del controlador cliente de cada DBMS, FireDAC se encargará de realizar la conversión a la codificación que corresponda. Análogamente los conjuntos de datos resultantes, por ejemplo los obtenidos

de una consulta, pasarán por el mismo proceso cuando sea aplicable. Lo único que hemos de tener en cuenta, por tanto, es que la configuración del controlador cliente que estemos utilizando sea la adecuada (véase siguiente apartado).

Dado que los controles utilizados para solicitar y mostrar información, con los que diseñamos nuestras interfaces de usuario, almacenan internamente los datos con codificación Unicode, uno de los pocos casos en los que necesitaremos realizar conversiones explícitas será cuando la información proceda de una fuente externa. Supongamos que vamos a importar o exportar datos de un archivo de texto a una base de datos, por ejemplo utilizando el componente `TFDDataMove`. Implícitamente se establecerá la codificación del conjunto de datos (clase derivada de `DataSet`) que actuará como origen o destinatario, pero la codificación del archivo de texto es necesario establecerla explícitamente mediante la propiedad `TextFileEncoding` del citado componente. Otro tanto ocurriría al utilizar el componente `TFDBatchMoveTextReader` para leer un archivo de texto o el componente `TFDBatchMoveTextWriter` para escribir en él.

Veamos la implementación práctica del caso que acaba de mencionarse. Concretamente tendremos unos datos en un archivo CSV externo con codificación UTF-8, datos que supuestamente queremos importar a una base de datos aunque, en este ejercicio, vamos a limitarnos a mostrarlos una vez que se encuentren en un `TFDQuery`, sin llegar a transferirlos realmente a un RDBMS. El contenido del archivo CSV con el que vamos a trabajar es el mostrado parcialmente en la Figura 8.12.

EJEMPLO 8.4 `UnicodeBDD.dproj`

Puedes encontrar este proyecto completo en la carpeta `UnicodeBDD`. No se precisa ninguna configuración adicional para poder compilar y ejecutar, salvo disponer el archivo `japeng.csv` en la carpeta adecuada.

El objetivo es importar el contenido del archivo CSV, usando la codificación adecuada, a un `TFDQuery` que, a su vez, conectaremos a una cuadrícula de datos a fin de mostrar la información. A continuación se detalla el proceso a seguir.

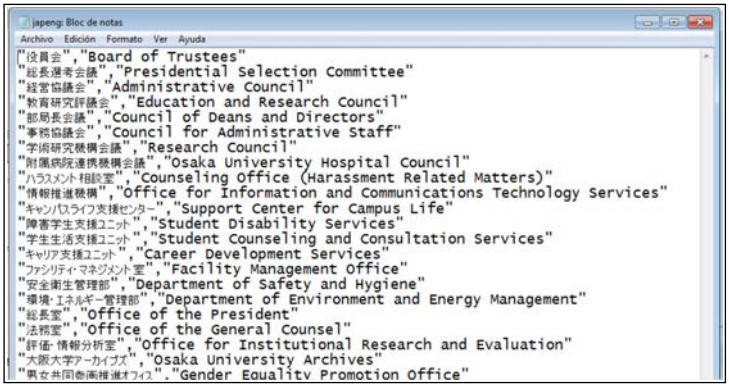


Figura 8.12 VISTA PARCIAL DEL CONTENIDO DEL ARCHIVO CSV

Partiendo de un proyecto vacío vamos a agregar un módulo de datos, como en los demás ejercicios, en el que incluiremos la siguiente lista de componentes:

- `TFDConnection`: Este componente servirá como punto de unión para los dos siguientes, facilitando el uso del motor de bases de datos de SQLite. Únicamente modificaremos la propiedad `LoginPrompt`, a la que asignaremos el valor `False`.
- `TFDLocalSQL`: Nos permitirá trabajar directamente con la base de datos en memoria, efectuando la consulta de recuperación de datos. Al agregarlo debería conectarse automáticamente con el anterior, algo que podemos verificar en el valor de la propiedad `Connection`.
- `TFDQuery`: Será el componente que ejecute la consulta y obtenga como resultado el conjunto de datos. Asignaremos a su propiedad `SQL` la consulta `SELECT * FROM japeng`.
- `TFDMemTable`: Este componente será el destinatario de la operación de transferencia de datos desde el archivo CSV, generando el `DataSet` que se transferirá al `TFDQuery` llegado el momento.
- `TFDBatchMoveTextReader`: Será el componente encargado de leer los datos almacenados en el archivo CSV, para lo cual es necesario establecer la configuración adecuada modificando las siguientes propiedades:

- `Encoding`: Seleccionaremos de la lista desplegable la opción `ecUTF8`, indicando así la codificación en que se encuentran los datos.
- `FileName`: Haremos clic en el botón asociado para abrir el cuadro de diálogo que nos permitirá elegir el archivo que deseamos abrir.
- `DataDef`: Mediante esta propiedad estableceremos el separador de campos y otros parámetros que afectan a la lectura de los datos. Asignando a la propiedad `RecordFormat` el valor `rfCommaDoubleQuote` obtenemos la configuración básica que necesitamos. Además usaremos la propiedad `Fields` para definir el nombre y tipo de los datos campos existentes en el archivo CSV, tal y como se muestra en la Figura 8.13.

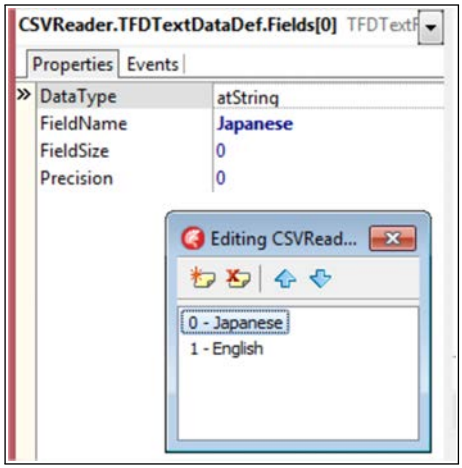


Figura 8.13 DEFINICIÓN DE LOS CAMPOS DEL ARCHIVO CSV

- `TFDBatchMoveDataSetWriter`: La finalidad de este componente será transferir los datos leídos por el anterior a un derivado de `TDataSet`. Para ello desplegaremos la lista asociada a su propiedad `DataSet` y elegiremos el `TFDMemTable` introducido anteriormente.
- `TFDBatchMove`: Es el último componente a agregar al módulo de datos. Se encargará de conectar a los dos anteriores mediante las propiedades `Reader` y `Writer`. La propiedad `Analyze` ha de quedar vacía, con todas

las opciones desmarcadas, ya que el `TFDBatchMoveTextReader` tiene la información necesaria para interpretar el contenido del archivo CSV.

Al ejecutar la aplicación se invocará al método `Open` del `TFDQuery`, lo cual provocará que este demande del `TFDLocalSQL` la ejecución de la consulta que asignamos a su propiedad `SQL`. Lo que hará este último componente será generar el evento `OnGetDataSet`, en el que introduciremos el código mostrado a continuación. El método `Execute` del `TFDBatchMove` se encargará de ejecutar la transferencia de datos del archivo CSV al `TFDMemTable`, resultado que finalmente será devuelto como contenido del `TFDQuery`:

```
1  procedure TdmUnicode.FDLocalSQL1GetDataSet(ASender: TObject;↙
       ↳  const ASchemaName,
2    AName: string; var ADataSet: TDataSet; var AOwned: Boolean↙
         ↳ );
3  begin
4    try
5      FDMemTable1.Close;
6      FDBatchMove1.Execute;
7      ADataSet := FDMemTable1;
8      AOwned := True;
9    except on E: Exception do
10     ShowMessage(E.Message);
11   end;
12 end;
```

Listado 8.10 Transferencia de los datos CSV al `TFDQuery`

En cuanto a la interfaz de usuario de la aplicación, únicamente agregaremos al formulario un control `TGrid`. Enlazaremos este con el componente `TFDQuery` del módulo de datos usando, como es habitual, el diseñador de **LIVEBINDINGS**. Por lo demás, únicamente usaremos el evento `OnShow` del formulario para invocar al método `Open` del `TFDQuery`, desencadenando así la sucesión de eventos que provocará la lectura del archivo CSV, su transferencia a la base de datos en memoria y, finalmente, la visualización en la cuadrícula. El resultado será similar al mostrado en la Figura 8.14 siempre y cuando la configuración de la codificación del archivo y el componente `TFDBatchMoveTextReader` sea

la adecuada. Si cambiásemos la propiedad `Encoding` de este último el texto obtenido probablemente sería ilegible.

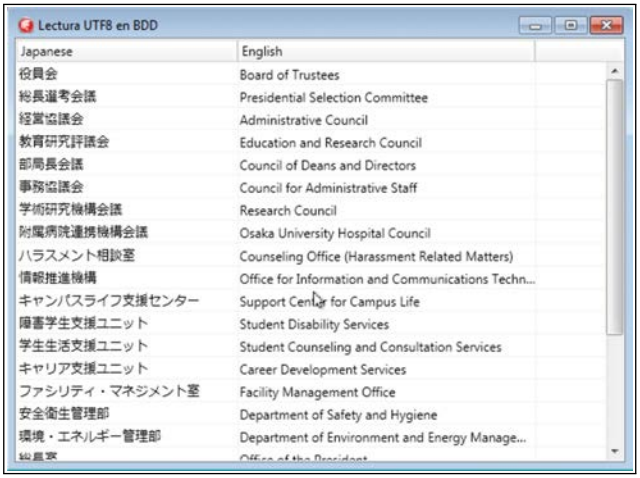


Figura 8.14 LA APLICACIÓN MOSTRANDO LOS DATOS LEÍDOS DEL ARCHIVO CSV.

> **NOTA**
>
> *Aunque en este ejercicio el archivo es CSV y se han obtenido un conjunto de filas y columnas de datos, la configuración al leer de un archivo externo también se aplicaría al recuperar el contenido de un archivo de texto, por ejemplo para almacenarlo en un campo de la base de datos. En este caso se introduciría en el campo únicamente el texto ya convertido (si fuese necesario), pero no tiene sentido incluir el BOM ya que ni los RDBMS ni la aplicación lo necesitan.*

8.3.2 Aspectos específicos de los RDBMS

Una vez que tenemos la información en nuestra aplicación, ya sea obtenida de una fuente externa como en el ejercicio previo o bien tras la introducción directa

por parte del usuario, llega el momento de almacenarla en la base de datos. Para evitar cualquier pérdida de información es necesario que el controlador cliente esté adecuadamente configurado, para lo cual podemos usar el cuadro de diálogo **CONNECTION EDITOR** asociado al componente `TFDConnection` (véase la Figura 8.15).

La configuración por defecto para determinados RDBMS ya asume el uso de UTF-8 como codificación para el texto, por lo que no se precisa ningún ajuste adicional. Si elegimos SQL Server como controlador de la conexión, por ejemplo, se asumirá que la codificación es Unicode. Al trabajar con Interbase, Firebird u Oracle, por el contrario, es nuestra responsabilidad configurar la codificación.

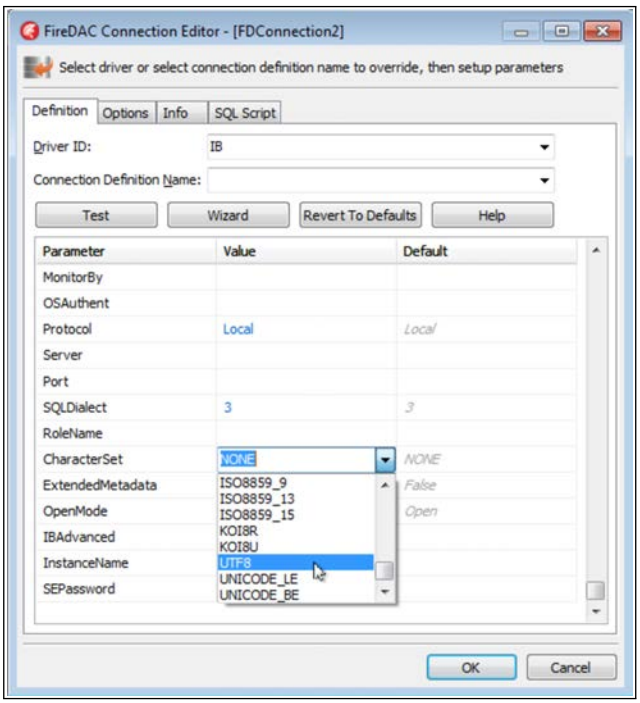


Figura 8.15 CONFIGURACIÓN DE LA CODIFICACIÓN PARA UNA CONEXIÓN A INTERBASE

Si creamos la base de datos utilizando una herramienta externa a Delphi, habremos de utilizar las opciones específicas de la misma para configurar la codi-ficación de caracteres. Posteriormente, al conectar con ella mediante FireDAC, será cuando fijemos la configuración del software cliente mediante el anterior cuadro de diálogo.

8.4 Resumen

En este capítulo hemos conocido los aspectos fundamentales de la codificación de caracteres con Unicode, frente a estándares heredados como ASCII, y la forma en que funcionan UTF-8, UTF-16 y UTF-32, incluyendo el uso de la cabecera BOM allí donde es necesario. Como se ha mostrado, el soporte de Unicode en Delphi es completo, incluyendo los componentes relacionados con acceso a bases de datos.

Se ha explicado cómo realizar conversiones entre codificaciones, cómo leer datos de archivos externos en distintas codificaciones y cómo trasladarlos a componentes FireDAC. A partir de ese punto, siempre y cuando el controlador de bases y el RDBMS estén apropiadamente configurados, no tendremos ningún problema en almacenar y recuperar información textual usando Unicode.

PARTE 2

ACCESO A DATOS EN APLICACIONES CLIENTE/SERVIDOR

En esta segunda parte del libro vamos a abordar los temas relacionados con el desarrollo de aplicaciones cliente/servidor, la arquitectura más conocida y utilizada para soluciones dirigidas a empresas e instituciones en las que existe un servidor de datos y un conjunto más o menos grande de clientes conectados al mismo, habitualmente mediante una infraestructura propia de comunicación como puede ser una red local.

En principio nos ocuparemos de los aspectos y procedimientos generales, los propios de FireDAC, en este tipo de configuración, para después abordar temas como la gestión de transacciones, estrategias de bloqueo o el trabajo sin conexión a la base de datos y uso de *cached updates*, para terminar tratando algunos detalles específicos del RDBMS InterBase.

Delphi permite conectar con un gran número de servidores distintos. El uso de algunos de ellos requerirá formación específica por parte de la persona que vaya a administrarlos. Aquí nos ceñiremos sobre todo a las tareas propias del desarrollador de aplicaciones, no a las del administrador de bases de datos.

Introducción al desarrollo cliente/servidor **263**

Transacciones, bloqueos y notificación de cambios **311**

Trabajar sin conexión al RDBMS **333**

Interbase **359**

Capítulo 9

INTRODUCCIÓN AL DESARROLLO CLIENTE/SERVIDOR

En los ejercicios desarrollados en la primera parte de este libro hemos utilizado gestores de bases de datos locales, como SQLite, Microsoft Acccess e IBLite, que operan directamente sobre la información en el mismo equipo que ejecuta la aplicación. La comunicación entre los componentes FireDAC y el gestor de datos es directa y, además, exclusiva, por lo que no suele darse ningún tipo de conflicto de acceso a los datos ni de actualización de los mismos.

La Figura 1.6 (véase página 54, en el primer capítulo) mostraba cómo al acceder a RDBMS como InterBase, SQL Server u Oracle la comunicación entre la aplicación y el servidor de datos se efectúa habitualmente mediante TCP/IP,

ya sea con una comunicación sin intermediarios o a través de controladores heredados de tipo ODBC o dbExpress. El RDBMS permitirá múltiples conexiones procedentes desde distintos orígenes, existiendo un acceso concurrente a los datos. No debe asumirse, por tanto, que se tiene un acceso exclusivo a los mismos, sino que este es compartido con otros usuarios y aplicaciones.

En este capítulo vamos a conocer algunos conceptos fundamentales sobre FireDAC a la hora de desarrollar aplicaciones cliente/servidor. Todos ellos son genéricos y, por tanto, se aplican con independencia de cuál sea el RDBMS concreto con el que vayamos a trabajar.

9.1 Conexión con el servidor de datos

Ya sabemos que el primer paso para operar con una base de datos, con independencia de que sea local o remota, consiste en establecer una conexión con la misma. Para ello recurriremos al componente `TFDConnection`, encargado de comunicarse con el software cliente del RDBMS en cuestión aportando la información necesaria: el nombre de la base de datos, la ruta en que se encuentra, las credenciales de acceso, etc.

Veamos cuáles son las diferencias en este campo entre lo que ya conocemos y la conexión con una base de datos en configuración cliente/servidor.

9.1.1 Ruta y nombre de la base de datos

Para conectar con una base de datos local, como las utilizadas en capítulos previos, se precisa el nombre del archivo y la ruta en que se encuentra almacenado en el sistema de archivos local. La composición de dicha ruta varía según el sistema operativo en que se ejecute la aplicación, tal y como se mostró en su momento. En general se asume que el archivo será abierto en exclusiva por una aplicación, pudiendo esta realizar cualquier tipo de operación sobre los datos sin mayores limitaciones.

La conexión con un RDBMS en una configuración cliente/servidor también demanda conocer el nombre de la base de datos, pero la ruta será sustituida por el nombre (o la dirección IP) de la máquina en la que está ejecutándose el servidor de datos. Habitualmente el RDBMS estará atendiendo de forma simultánea a un

cierto número de clientes, por lo que, dependiendo de la capacidad de la línea de comunicación y el hardware del servidor, la conexión podría no ser inmediata, sino tardar un cierto tiempo o incluso fallar por completo en caso de que la comunicación se interrumpa. Estos son aspectos que, por regla general, no es necesario considerar en una configuración local.

> **NOTA**
>
> *Aunque es habitual que los servidores de datos y aplicaciones en una red propia tengan asignada una dirección IP estática, en lugar de obtenerla mediante DHCP (Dynamic Host Configuration Protocol), es recomendable no embeberla en el código de la aplicación sino mantenerla en un archivo de configuración externo, de forma que pueda modificarse fácilmente.*

Cada servidor de datos utiliza un puerto por defecto para aceptar las solicitudes de conexión entrantes por parte de los clientes. Estos, por tanto, han de enviar sus peticiones a dicho puerto. Cada controlador usará por defecto dicho puerto, pero si este se hubiese cambiado en la configuración del servidor también habrá que aplicar dicha modificación en la configuración del cliente. Esto implica normalmente un cambio en el componente `TFDConnection` o el archivo de configuración asociado. Lógicamente el citado puerto ha de encontrarse abierto para recibir conexiones entrantes en el servidor y salientes en los clientes. Un fallo habitual al comenzar a trabajar con esta configuración es no caer en la cuenta de que el software cortafuegos está bloqueando dichas conexiones, lo cual hace imposible la comunicación y, por tanto, la conexión con el RDBMS.

9.1.2 Autenticación de acceso

Una aplicación que almacena y recupera la información en el mismo sistema en que se ejecuta, usando una base de datos local, normalmente no requerirá credenciales adicionales. Se asume que si el usuario tiene acceso al sistema y puede ejecutar la aplicación, esta puede operar sobre los datos. El escenario en una configuración cliente/servidor es totalmente distinto. El servidor de datos normalmente dará servicio a múltiples aplicaciones, alojando información a la que no debe garantizarse un acceso universal para todos los usuarios.

Dependiendo del RDBMS que se utilice, y la forma en que esté configurado, nos podemos encontrar fundamentalmente con dos modalidades distintas de autenticación:

- **OS**: También conocida como autenticación de tipo *trusted*. Consiste en utilizar las credenciales con las que el usuario ha iniciado sesión en el sistema operativo para acceder al servidor de datos.
- **DBM**: Habitualmente denominada autenticación nativa, se basa en la configuración de los usuarios y sus contraseñas en la propia base de datos, de forma que es necesario aportar dichas credenciales a la hora de acceder al servidor.

Ciertos RDBMS, por ejemplo la versión 2.1 de Firebird, contemplan el uso de una tercera configuración de autenticación conocida como *mixed*. Si al conectar con el RDBMS se facilita un usuario y contraseña se recurre a la autenticación nativa, utilizándose la del sistema operativo en caso contrario.

> **NOTA**
>
> *Habitualmente la autenticación de tipo SO está asociada al uso de Windows como sistema operativo. En una red Windows los usuarios al iniciar sesión se identifican con unas credenciales que controla un servidor, un equipo con Windows Server, que determina qué operaciones y privilegios tiene cada usuario. Serían esas mismas credenciales las que se entregarían al RDBMS, por ejemplo SQL Server, para que este hiciese lo propio sobre los datos.*

El componente `TFDConnection` cuenta con tres propiedades que determinan qué método de autenticación se usará y las credenciales a aportar en caso necesario. Son la siguientes:

- `OSAuthent`: Esta propiedad booleana determina si se utilizará la autenticación de tipo OS, asignándole el valor `True`, o la del propio RDBMS, con el valor `False`. Este último es el valor que se asume por defecto.
- `UserName` y `Password`: En caso de que la anterior propiedad sea `False` y la conexión al servidor de datos requiera autenticación, estos dos campos aportarán las credenciales necesarias.

En el Editor de conexión del componente `TFDConnection` encontraremos la propiedad `OSAuthent` (véase la Figura 9.1) únicamente para ciertos controladores de bases de datos, como InterBase, Firebird y Microsoft SQL Server. Habitualmente la autenticación con credenciales del SO solo suele utilizarse con SQL Server.

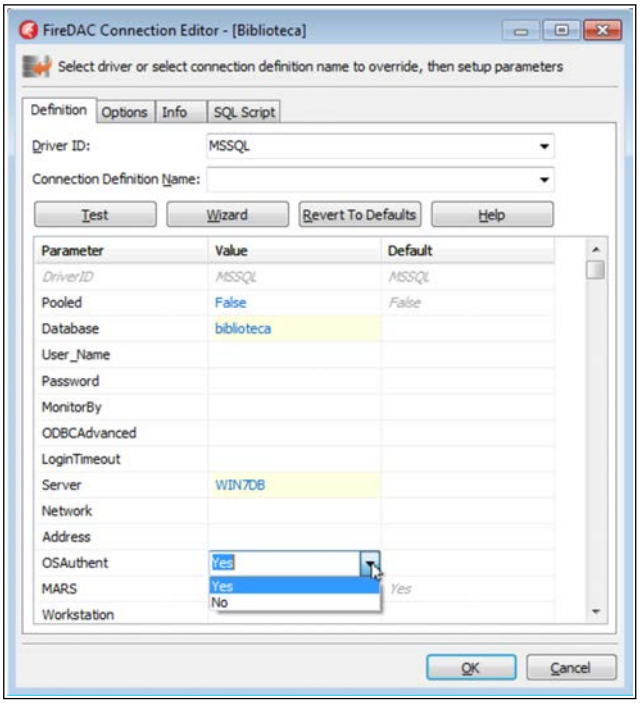


Figura 9.1 CONFIGURACIÓN DEL MÉTODO DE AUTENTICACIÓN

9.1.3 Reutilización de conexiones

El proceso de conexión con una base de datos remota, según se apuntaba anteriormente, es un proceso costoso en tiempo y recursos, al implicar el intercambio de una sucesión de paquetes de información a través de una red de comunicaciones. En caso de que una aplicación vaya a abrir varias conexiones

simultáneas con la misma base de datos, por ejemplo en múltiples hilos de ejecución (*threads*) en paralelo, ese coste se multiplica. Para evitar este efecto, FireDAC incorpora un mecanismo de reutilización de conexiones que es necesario activar explícitamente, ya que por defecto no lo está.

La reutilización de conexiones consiste en mantener un grupo o *pool* de conexiones ya preparadas y abiertas, de forma que cuando la aplicación requiere una conexión, por ejemplo para ejecutar una consulta, se toma una de las disponibles en el grupo que no esté actualmente en uso. Cuando la ejecución de la consulta finaliza, la conexión se devuelve al *pool* estando de nuevo disponible. De esta forma no se abren y cierran conexiones con el RDBMS de forma reiterada, reduciendo el tiempo necesario para realizar cada tarea. Al mismo tiempo, al mantener un grupo limitado de conexiones abiertas con el servidor, también se reduce la carga en este respecto a un escenario en el que cada hilo mantuviese abierta una conexión independiente.

Para activar la reutilización de conexiones configuraremos el componente `TFDConnection` atendiendo a las dos directrices siguientes:

1. **Configuración compartida**: Para poder reutilizar una conexión es necesario que los parámetros de configuración de los distintos casos de uso sean los mismos. En la conexión básicamente se aportará el nombre de la base de datos sobre la que se operará y las credenciales necesarias para operar sobre la misma, nada más.
2. `Pooled`: Entre los parámetros de configuración encontraremos esta propiedad booleana, a la que hemos de dar el valor `True` para activar la reutilización de la conexión.

> **NOTA**
>
> *La reutilización o pooling solo puede utilizarse en conexiones persistentes, alojadas en un archivo propio de conexiones del proyecto, o bien en conexiones privadas a la aplicación, configuradas en tiempo de ejecución.*

El funcionamiento del *pool* de conexiones compartidas puede configurarse utilizando los siguientes parámetros de la propiedad `Params` del componente `TFDConnection`:

- `PoolMaximumItems`: Esta propiedad establece el número máximo de conexiones que se almacenarán en el *pool*. Por defecto toma el valor 50 que, en general, será más que suficiente para la mayoría de los proyectos. El número de conexiones disponibles en el *pool* irá descendiendo a medida que la aplicación vaya demandándolas para trabajar sobre la base de datos, volviendo a incrementarse a medida que dichas tareas finalicen y las conexiones se devuelvan. Si en algún momento el *pool* está vacío y la aplicación solicita una nueva conexión se producirá una excepción.

- `PoolExpireTimeout`: Establece el tiempo en milisegundos tras el cual una conexión inactiva puede ser cerrada y eliminada del *pool*. Por defecto su valor es `90 000`, lo que implica que aquellas conexiones que lleven más de 90 segundos sin ser utilizadas se cerrarán y eliminarán.

- `PoolCleanupTimeout`: Tiempo en milisegundos que ha de transcurrir, adicional al establecido por la propiedad anterior, antes de que FireDAC elimine las conexiones no utilizadas. La configuración por defecto es de 30 segundos.

Un `TFDConnection` configurado como acaba de describirse tomará una conexión del *pool* cada vez que se invoque al método `Open` o se asigne el valor `True` a la propiedad `Connected`, por ejemplo al abrir una tabla o ejecutar una consulta. Al cerrar dicha conexión, volviendo a dar el valor `False` a `Connected`, la conexión en realidad no se cerrará, sino que se devuelve al *pool* a fin de poder reutilizarla. Si transcurrido el tiempo establecido por las propiedades anteriores el programa no ha precisado una conexión disponible en el *pool*, esta se cierra y se destruye. De esta forma, el número real de conexiones existentes en el *pool* irá adaptándose a las necesidades de la aplicación en cada momento, si bien nunca podrá superar el número establecido por la propiedad `PoolMaximumItems`.

NOTA

Para aplicaciones con conexión esporádica al RDBMS, capaces de operar offline la mayor parte del tiempo, véase el Capítulo 11.

9.1.4 Gestión de las conexiones durante la ejecución

Al ejecutar una aplicación que utiliza FireDAC para acceder a bases de datos se crea una instancia de la clase `TFDCustomManager`, a la que podemos acceder mediante la función `FDManager`, encargada de leer las definiciones de conexiones existente en el archivo de configuración, agregar nuevas conexiones, mantener los datos de las conexiones abiertas, gestionar el *pool* de conexiones reutilizables descrito en el apartado anterior, etc. Las propiedades y métodos ofrecidos por dicha clase pueden sernos de utilidad en distintos escenarios. Aquí únicamente se mencionan algunos de ellos.

Mediante la propiedad `Connections` es posible acceder a la lista de conexiones actualmente abiertas. La propiedad `ConnectionCount` indica cuántos elementos hay en dicha lista. Cada elemento es un objeto `TFDCustomConnection` con los datos de la conexión y métodos que permiten actuar sobre la misma.

La información relativa a definiciones de conexiones persistentes puede recuperarse mediante los métodos `GetConnectionDefNames`, que devuelve una lista con los nombres de cada conexión, y `GetConnectionDefParams`, encargado de facilitar la lista de parámetros de una conexión concreta. Asimismo pueden agregarse nuevas definiciones de conexiones, mediante el método `AddConnectionDef`, así como eliminar cualquiera de las existentes, con el método `DeleteConnectionDef`.

Las conexiones activas para una cierta definición de conexión pueden cerrarse usando el método `CloseConnectionDef`. Este precisa como parámetro el nombre asignado a la conexión (o el facilitado como primer parámetro al método `AddConnectionDef`). Este método puede utilizarse, por ejemplo, para cerrar todas las conexiones existente en el *pool* de conexiones reutilizables asociadas a una definición concreta, sin esperar a los tiempos de expiración establecidos.

Veamos en la práctica cómo usar dos de los métodos citados para obtener información de las conexiones FireDAC definidas en el sistema. Para ello construiremos una aplicación muy simple, con una interfaz de usuario conteniendo dos controles `TListBox`. El primero estará alineado al margen izquierdo del formulario y mostrará los nombres de las conexiones existentes. El segundo ocupará el registro del formulario y, al seleccionar una conexión del anterior, informará sobre los parámetros de configuración.

Como es habitual, comenzaremos iniciando un nuevo proyecto de tipo multi-dispositivo. A continuación incluiremos los dos controles `TListBox`, modificando su propiedad `Align`. La primera tomará el valor `Left` y la segunda el valor `Client`. Acto seguido hacemos doble clic sobre el formulario, para abrir el método asociado al evento `OnCreate` que será el que usemos para llenar la primera lista. Para ello usaremos el método `GetConnectionDefNames` del objeto devuelto por `FDManager`, tal y como se aprecia en el listado siguiente. Al seleccionar un elemento de esta lista se generará el evento `OnChange`, señal que aprovecharemos para recuperar la lista de parámetros de la conexión indicada y mostrarla en la segunda lista.

EJEMPLO 9.1 `CSConnection.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSConnection`. No se precisa ninguna configuración adicional para poder compilar y ejecutar.

```
1 procedure TfrmMain.FormCreate(Sender: TObject);
2 begin
3   FDManager.GetConnectionDefNames(ListBox1.Items);
4 end;
5
6 procedure TfrmMain.ListBox1Change(Sender: TObject);
7 begin
8   FDManager.GetConnectionDefParams(ListBox1.Selected.Text, ↙
      ↳ ListBox2.Items);
9 end;
```

Listado 9.1 Introducción de datos en las listas

La Figura 9.2 muestra el programa en funcionamiento, con una de las conexiones predefinidas seleccionada y sus parámetros de conexión a la derecha.

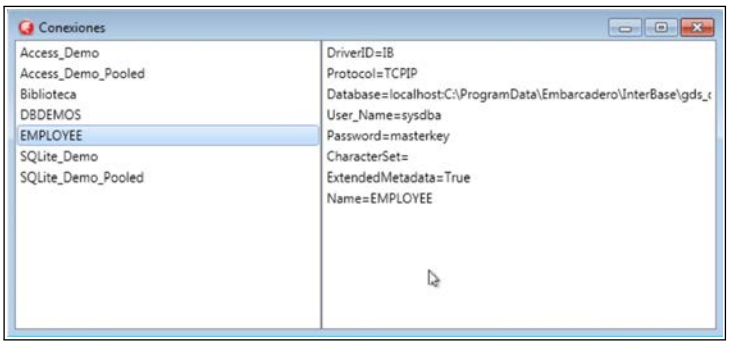


Figura 9.2 LISTA DE CONEXIONES Y PARÁMETROS

9.2 Obtención de datos

Una vez que se haya establecido la conexión con el servidor de datos, la recuperación de información se lleva a cabo utilizando los mismos componentes y técnicas descritos en los capítulos previos. Existen, no obstante, algunas directrices que es recomendable seguir. La más inmediata es que, salvo excepciones, nunca debería utilizarse el componente `TFDTable`, ya que raramente nos interesará transferir desde el servidor hasta los clientes todas las filas y columnas de una tabla. Por ello en aplicaciones cliente/servidor el método preferente para obtener datos del RDBMS es el componente `TFDQuery` o, en su defecto, el componente `TFDStoredProc`.

En una configuración cliente/servidor el cliente, la aplicación que actúa como interfaz de usuario, ha de solicitar al servidor los datos que deben mostrarse al usuario en cada momento. El número de filas y columnas a transferir desde el servidor a través de la red tendrá un importante impacto en la experiencia del usuario con nuestro programa, pudiendo introducir demoras que deterioren dicha experiencia.

Los apartados de esta sección del capítulo sugieren el uso de algunas técnicas básicas dirigidas a limitar la cantidad de datos a transferir entre cliente y servidor, reduciendo el tiempo de respuesta de la aplicación y el uso de recursos como el ancho de banda, la memoria en el cliente o la entrada/salida a disco en el

servidor. Conjuntamente, estas técnicas mejorarán la escalabilidad de la solución y también la experiencia del usuario con la misma.

9.2.1 Selección de columnas

Por regla general las tablas de una base de datos contienen muchos más atributos que los que es necesario mostrar en la interfaz de usuario de la aplicación. Por esta razón, las consultas del tipo `SELECT * FROM` deberían evitarse, especificando en su lugar las columnas concretas que es necesario recuperar. Esta es una acción que tiene un efecto multiplicativo, ya que cada columna eliminada de la consulta afecta a todas las filas resultantes de la misma. El resultado es que el RDBMS ha de recuperar menos información del disco y, sobre todo, se reduce el volumen de datos a transportar por la red.

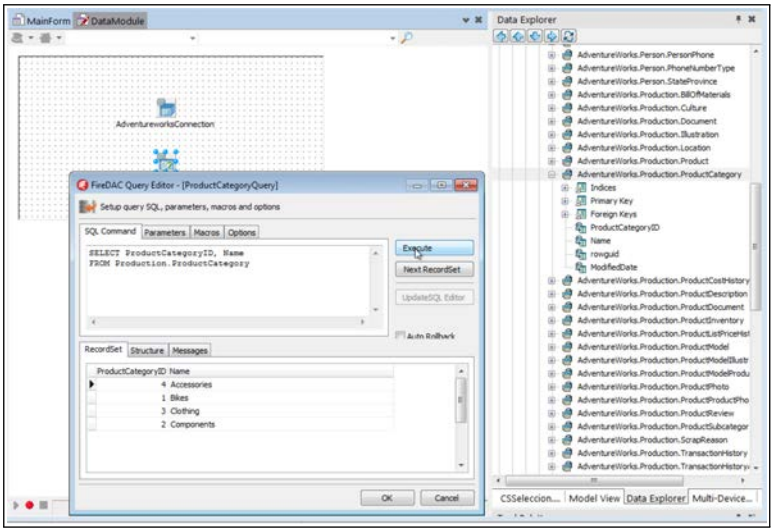


Figura 9.3 SELECCIÓN DE CAMPOS CONCRETOS EN UNA CONSULTA

Si trabajamos habitualmente con el **DATA EXPLORER** ya sabemos que al seleccionar una tabla del mismo y arrastrarla al módulo de datos lo que se obtiene es un componente `TFDTable`, configurado para recuperar todo el contenido de

la tabla. En la práctica, cada uno de estos componentes debería ser sustituido por un `TFDQuery`, utilizando el Editor de consultas asociado para seleccionar exactamente los campos que se precisan. La lista de campos de cada tabla es algo que podemos consultar en el propio **DATA EXPLORER**, como se aprecia en la Figura 9.3. Esta herramienta, el Editor de consultas, también nos permite ejecutar la consulta y comprobar el resultado obtenido.

Habitualmente, además de las columnas de datos propiamente dichas también será necesario seleccionar aquellas que identifiquen de manera única cada una de las filas. En la Figura 9.3, por ejemplo, solamente se quiere mostrar el nombre de cada una de las categorías de productos existentes, pero también se ha incluido en la consulta la columnas con el identificador de dichas categorías. Este dato será necesario para, por ejemplo, referenciar una cierta categoría desde otra tabla o actualizar el nombre de una categoría.

> **NOTA**
>
> *La base de datos que aparece abierta en el* **DATA EXPLORER** *en la Figura 9.3 es* `AdventureWorks`*. Se trata de una base de datos de ejemplo ofrecida por Microsoft para SQL Server. Puedes descargarla desde la web de Microsoft, desempaquetarla y definir una conexión FireDAC para acceder a la misma. Será la base de datos que utilicemos en varios de los ejemplos de esta parte del libro.*

9.2.2 Filtrado de filas

El mismo razonamiento aplicable a las columnas de una tabla es extrapolable también a sus filas. Normalmente nunca ejecutaremos una consulta sobre la base de datos sin incluir una cláusula `WHERE` a fin de recuperar exclusivamente aquellas filas que necesite la aplicación, en lugar de transportar toda la tabla completa desde el servidor al cliente.

Esto es especialmente importante en tablas con mucha información. No es el caso de la tabla `ProductCategory` usada a modo de ejemplo en el apartado anterior, ya que cuenta con únicamente cuatro filas. Tomemos en su lugar la tabla `SalesOrderDetail` de la misma base de datos. Contiene más de 120 000 filas de datos correspondientes a líneas de pedido, por lo que una consulta típica

del tipo `SELECT * FROM Sales.SalesOrderDetail` tardaría un tiempo importante en ejecutarse al tener que transferir dicho volumen de datos por la red. Además, el programa que actúa como cliente deberá contar con memoria necesaria para almacenar toda esa información.

En el caso concreto que se expone el filtrado sería muy obvio: la aplicación solamente requerirá las líneas correspondientes a un pedido concreto. La selección de columnas y filtrado de filas permiten recuperar exactamente los datos que se precisan, como se muestra en la Figura 9.4. En este caso el filtro de filas utiliza un parámetro sustituible, de forma que la consulta se podría utilizar de forma genérica para cualquier pedido.

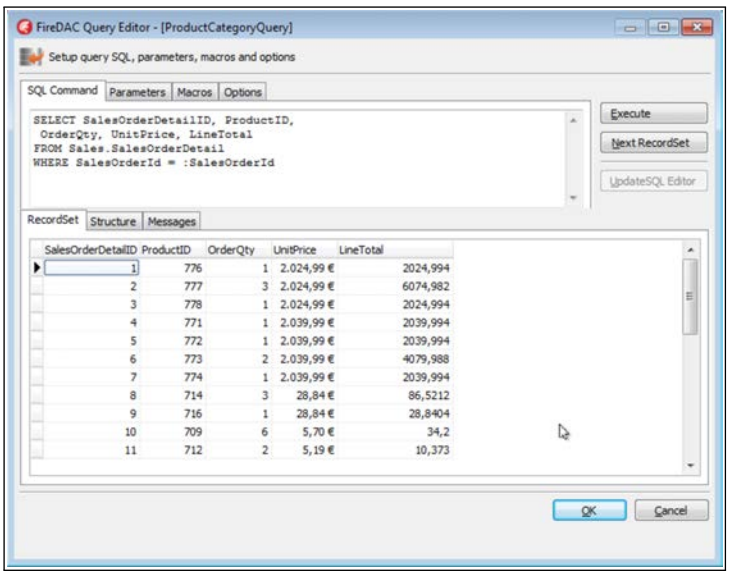


Figura 9.4 CONSULTA CON FILTRADO DE FILAS Y SELECCIÓN DE COLUMNAS

9.2.3 Paginación de resultados

Con independencia del filtrado que llevemos a cabo en la propia consulta, la configuración por defecto del componente `TFDQuery` limita el número de filas

que son transmitidas desde el servidor al cliente. Esta configuración por defecto, ajustada para ir paginando los resultados a medida que se precise, puede modificarse cambiando los valores de los campos de la propiedad `FetchOptions` del citado componente.

Las subpropiedades esenciales de `FetchOptions` en relación a la página de resultados son las siguientes:

- `RowsetSize`: Establece el número de filas que serán transferidas en cada operación de recuperación o, desde otra perspectiva, fija el tamaño de cada conjunto de filas o *recordset*. El valor por defecto es 50. un `TFDQuery` mantendrá normalmente en memoria dos conjuntos de filas de dicho tamaño, actuando como una ventana deslizable sobre el total de filas resultantes de la consulta. La recuperación de paquetes adicionales dependerá de la siguiente propiedad.

- `Mode`: Esta propiedad indica al componente cómo se gestionará la recuperación de paquetes de filas desde el servidor. Los valores que puede tomar son:

  - `fmOnDemand`: Es el valor por defecto. Los conjuntos de filas son solicitados al servidor a medida que lo demande la navegación por los datos. FireDAC utilizará en el servidor el recurso más apropiado para facilitar este tipo de funcionamiento, según el RDBMS utilizado.
  - `fmAll`: Con esta configuración el componente recuperará todas las filas resultantes de la consulta en cuando se abra la conexión, almacenándolas localmente. Este es un proceso que puede tardar un cierto tiempo, dependiendo de la cantidad de datos a transferir, pero una vez completado permite navegar por la información sin necesidad de solicitudes ulteriores.
  - `fmManual`: Al asignar este valor a `Mode` la responsabilidad de solicitar los paquetes de datos al servidor recae en el programador, que deberá agregar el código necesario para efectuar dicha tarea.

De optarse por el modo de recuperación de datos manual, asignando el valor `fmManual` a `FetchOptions.Mode`, al abrir el `TFDQuery` no se recuperarán filas de forma automática. Tendremos que utilizar los métodos `FetchAll` y `FetchNext` para solicitar todas las filas resultantes de la consulta o el siguiente

paquete, respectivamente. El número máximo de filas de cada paquete vendrá determinado por el valor de la propiedad `RowsetSize` descrita anteriormente.

Las filas a incluir en cada paquete de datos, con independencia de que su recuperación sea manual o automática, también se verá afectada por el valor de las propiedades `RecsSkip` y `RecsMax` de la propiedad `FetchOptions`. Por defecto la primera tiene el valor 0 y la segunda el valor -1, por lo que se recuperará desde la primera fila un paquete de datos del tamaño indicado por `RowsetSize`. Asignando a `RecsSkip` un valor distinto de cero conseguiremos *saltar* el número de filas indicadas antes de comenzar a recuperar. Con `RecsMax` limitaremos el número de filas a obtener, indicando el valor -1 la ausencia de límite.

> **NOTA**
>
> *Si tras asignar el valor `fmManual` a `FetchOptions.Mode` abrimos el Editor de consultas del `TFDQuery` para ajustar la consulta o comprobar su funcionamiento, hemos de tener en cuenta que al ejecutar la consulta no se obtendrá ningún dato. Tendríamos que cambiar temporalmente el valor a `fmOnDemand` para que esta herramienta del entorno funcione como es habitual.*

Veamos en la práctica cómo servirnos de dos de las propiedades citadas, `RecsSkip` y `RecsMax`, para producir un efecto de paginación manual en la interfaz de usuario. Para ello, partiendo de un nuevo proyecto al que agregamos un módulo de datos, reproduciremos los pasos descritos a continuación:

1. Agregamos al módulo de datos un `TFDConnection` configurado para acceder a la base de datos `AdventureWorks` de SQL Server. Si no contamos con esta podemos utilizar cualquier otra, incluso trabajando con un RDBMS distinto, en cuyo caso habría que cambiar la consulta y referencias a datos en los pasos siguientes.

2. Introducimos un componente `TFDQuery` en el módulo de datos, conectándolo con la conexión anterior. A continuación abrimos el Editor de consultas asociado y escribimos la sentencia SQL mostrada en el Listado 9.2.

```
1 SELECT SalesOrderId, OrderDate, CustomerID, TotalDue
2 FROM Sales.SalesOrderHeader
```

Listado 9.2 Consulta de recuperación de pedidos

3. Cambiamos al formulario que actuará como interfaz de usuario e insertamos en él los componentes mostrados en la Figura 9.5. El control `TGrid` está vinculado con el `TFDQuery`, de forma que mostrará los datos recuperados por este. El `TEdit` inferior tiene a `True` su propiedad `ReadOnly`, por lo que no podrá ser modificado en ejecución.

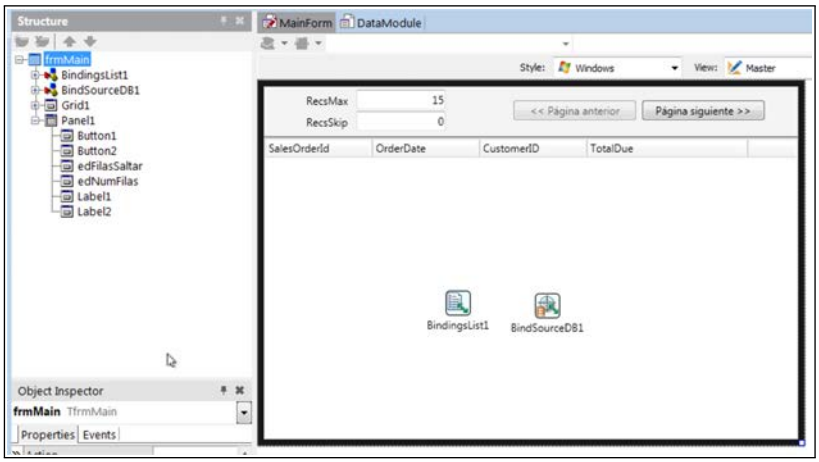


Figura 9.5 ELEMENTOS A INTRODUCIR EN EL FORMULARIO

4. Abrimos el módulo de código asociado al formulario, agregando a la sección privada de la definición de la clase la declaración de una variable entera y un procedimiento. La variable, llamada `UltimaFila`, nos servirá para establecer el número de filas a saltar en cada operación de lectura.

```
1   private
2     UltimaFila: Integer;
3     procedure Fetch;
```

Listado 9.3 Miembros a añadir a la definición de clase del formulario

5. A continuación escribimos la implementación del procedimiento `Fetch`, cuya finalidad será provocar una solicitud de datos al servidor con los parámetros adecuados. Dado que el modo de recuperación de datos (propiedad `FetchOptions.Mode`) del `TFDQuery` no se ha modificado, bastará con desconectar y volver a conectar con la base de datos para provocar una solicitud de recuperación de filas. Las filas concretas a obtener vendrán determinadas por el valor de las propiedades `RecsSkip`, que establecemos a partir de la variable `UltimaFila`, y `RecsMax`, valor que recuperamos del primer `TEdit` existente en la interfaz. La actualización del contenido del `TGrid` será automática, tras lo cual procedemos a controlar el estado de los botones dependiendo de que nos encontremos en la primera fila o en la última.

```
1  procedure TfrmMain.Fetch;
2  begin
3    with dmPaginate.SalesorderheaderQuery do
4    begin
5      if UltimaFila < 0 then
6        UltimaFila := 0;
7
8      Disconnect;
9      FetchOptions.RecsSkip := UltimaFila;
10     FetchOptions.RecsMax := StrToInt(edNumFilas.Text);
11     Open;
12
13     Button1.Enabled := UltimaFila <> 0;
14     Button2.Enabled := RecordCount >= StrToInt(edNumFilas↙
           ↳ .Text);
15     edFilasSaltar.Text := IntToStr(UltimaFila);
16   end;
17 end;
```

Listado 9.4 Implementación del procedimiento Fetch

6. Al anterior método se realizarán llamadas desde tres puntos diferentes: el método asociado al evento `OnShow` del formulario y los métodos vinculados al evento `OnClick` de los botones. En los tres casos primero se actualiza el contenido de la variable `UltimaFila`, primero, y se invoca

al método `Fetch`, a continuación. El código correspondiente a esos tres métodos es el siguiente:

```
procedure TfrmMain.FormShow(Sender: TObject);
begin
  UltimaFila := 0;
  Fetch;
end;

procedure TfrmMain.Button1Click(Sender: TObject);
begin
  UltimaFila := UltimaFila - StrToInt(edNumFilas.Text);
  Fetch;
end;

procedure TfrmMain.Button2Click(Sender: TObject);
begin
  UltimaFila := UltimaFila + StrToInt(edNumFilas.Text);
  Fetch;
end;
```

Listado 9.5 Código asociado a los botones

EJEMPLO 9.2 `CSPaginate.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSPaginate`. Deberás ajustar el componente `TFDConnection` para conectar con el servidor SQL Server y abrir la base de datos `AdventureWorks`.

Al ejecutar el programa la cuadrícula debería mostrar tantas filas como valor por defecto se haya asignado al primer `TEdit`. Una vez en funcionamiento podemos cambiar dicho valor, así como utilizar los botones para ir navegar por los datos. En la Figura 9.6 se muestra el programa en ejecución.

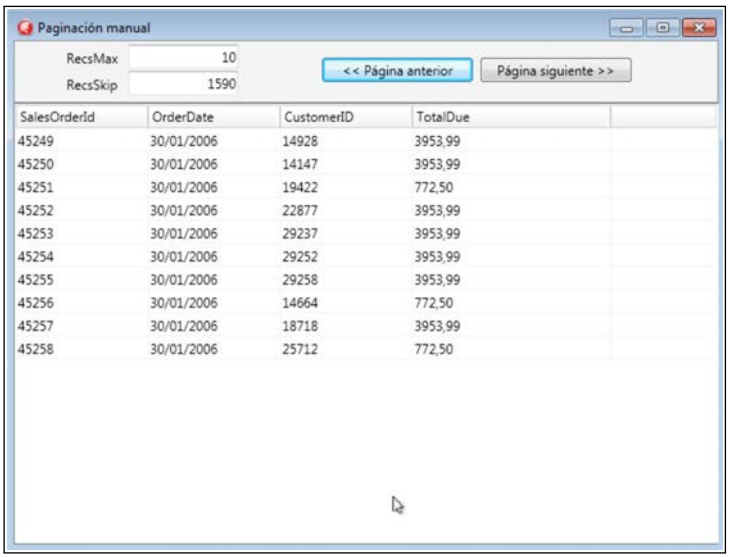


Figura 9.6 EL PROGRAMA EN FUNCIONAMIENTO MOSTRANDO UNA PÁGINA DE DATOS

9.3 Edición de datos

En las secciones previas de este capítulo nos hemos ocupado de la conexión de una aplicación cliente con el servidor de datos y de la recuperación de información, transfiriendo filas de datos desde el RDBMS a la interfaz del proyecto. Habitualmente dicha interfaz también permitirá al usuario manipular dichos datos, ya sea añadiendo nuevas filas, cambiando valores de las existentes o eliminándolas.

La finalidad de esta sección del presente capítulo es abordar diversos temas relacionados con la edición de datos en un entorno cliente/servidor, siempre centrados en el trabajo con componentes FireDAC. Algunos de estos temas, como las actualizaciones por lotes, serán extendidos posteriormente, en los siguientes capítulos.

9.3.1 Cómo actualiza FireDAC los datos

Los proyectos que hemos desarrollado como ejemplo hasta ahora, especialmente en la primera parte del libro, se caracterizan por facilitar la actualización inmediata de los datos que están mostrándose. No hay más que actuar sobre los controles dispuestos en la interfaz de usuario, el `TGrid` o las opciones de navegación ofrecidas por el componente `TBindNavigator`, para agregar nuevas filas, cambiar el contenido de las existentes o eliminarlas. Estas operaciones sobre el conjunto de datos, representado por el componente `TFDTable`, `TFDQuery` o similar, se comunican de manera inmediata a la base de datos subyacente, con independencia de la naturaleza de esta.

El procedimiento que sigue FireDAC para determinar si una cierta operación es posible y, en caso afirmativo, enviarla al gestor de datos, consta de los pasos siguientes:

1. Se determina si el conjunto de datos es de solo lectura o no, consultando para ello la subpropiedad `ReadOnly` de la propiedad `UpdateOptions`. En caso negativo se comprueba si contempla la operación a llevar a cabo leyendo las propiedades booleanas `EnableInsert`, `EnableUpdate` y `EnableDelete`. Las tres son también miembros de la mencionada propiedad `UpdateOptions` (véase la Figura 9.7). Mediante estas cuatro propiedades podemos configurar un `TFDQuery` para que no permita ninguna operación de actualización, dando el valor `False` a `ReadOnly`, o bien desactivando acciones concretas como la inserción, actualización o borrado. Por defecto `ReadOnly` tendrá el valor `False` y las otras tres propiedades estarán a `True`.

2. Para operaciones de inserción y actualización es necesario activar en el conjunto de datos el modo apropiado antes de comenzar a asignar valores a los campos. Con este fin se recurre a los métodos `Append` y `Edit`, respectivamente. Tras haber dado a los campos el contenido deseado, una llamada al método `Post` se encarga de enviar al gestor de datos la sentencia SQL adecuada para completar la operación. Antes se llevarán a cabo comprobaciones, dependiendo del valor de las propiedades `CheckRequired`, `CheckUpdatable`, etc., de la propiedad `UpdateOptions`.

3. Para operaciones de borrado se invoca directamente al método `Delete` del conjunto de datos. Este se encargará de realizar las comprobaciones nece-

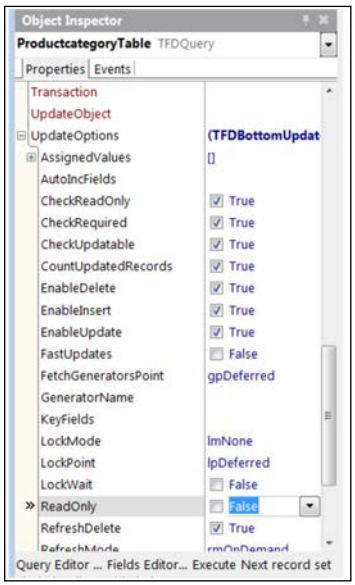


Figura 9.7 OPCIONES DE ACTUALIZACIÓN DE UN CONJUNTO DE DATOS

sarias, por ejemplo aplicando cambios que hubiese pendientes de confirmar, antes de proceder al borrado de la fila actual del conjunto de datos. Es una operación que provoca el avance a la fila siguiente, ya que la que era actual deja de existir.

En realidad los cambios producidos por los métodos `Post` y `Delete` serán enviados de inmediato al gestor de datos solo si la propiedad `CachedUpdates` del `TFDQuery` está a `False`. Ese es su valor por defecto y, dado que hasta el momento no lo hemos modificado en ninguno de nuestros proyectos, las operaciones de actualización de datos tienen un efecto inmediato.

> **NOTA**
>
> *Véase la Sección 11.2 del Capítulo 11 para saber más sobre la propiedad* `CachedUpdates` *y el uso de las actualizaciones de datos por lotes.*

9.3.2 Generación de las sentencias para actualizar datos

Las operaciones de inserción, modificación y borrado de datos son las tres que conllevan una actualización del contenido de la base de datos, debiendo traducirse en las correspondientes sentencias SQL de tipo `INSERT`, `UPDATE` y `DELETE`, respectivamente. Estas son generadas automáticamente por FireDAC, siempre que se cumplan unas ciertas premisas, y enviadas al RDBMS para su ejecución.

Lo primero que ha tenerse en cuenta es que solamente pueden generarse sentencias de actualización a partir de un `TFDQuery` que contiene una consulta, es decir, una sentencia `SELECT` de SQL. Dicha consulta deberá incluir entre los campos obtenidos aquellos que formen la clave primaria correspondiente a la tabla referenciada en la cláusula `WHERE`. FireDAC es capaz de identificar las columnas de dicha clave y utilizarla adecuadamente para limitar las filas afectadas. De no incluirse la clave primaria aún se intentará componer la sentencia de actualización, pero esta podría tener efectos indeseados si no se controla adecuadamente.

Actualización y clave primaria

A la hora de determinar las filas de una tabla que se verían afectadas por una operación de actualización podemos recurrir a tres enfoques diferentes. El que se utilice en cada momento dependerá del valor que se asigne a la propiedad `UpdateOptions.UpdateMode`. Los valores posibles son:

- `upWhereKeyOnly`: Este es el valor asignado por defecto. Establece que en la cláusula `WHERE` solamente aparecerán las columnas que formen la clave primaria.
- `upWhereChanged`: Seleccionando este modo se incluirán en la cláusula `WHERE` tanto las columnas de la clave primaria como aquellos cuyo contenido se haya modificado. De esta forma la aplicación se asegura de no cambiar o eliminar una fila que haya sido alterada por otro usuario desde que la recuperamos del servidor.
- `upWhereAll`: Asignando este valor a `UpdateMode` conseguiremos que en la cláusula `WHERE` se incluyan todas las columnas de la tabla, compa-

rando los valores que tenemos en la aplicación con los almacenados actualmente en la base de datos.

Comprobar el efecto de la propiedad `UpdateMode` sobre las sentencias de actualización es relativamente sencillo. No tenemos más que recurrir a la utilidad de monitorización de FireDAC para examinar las sentencias enviadas al servidor en cada caso. Para ello necesitaremos un programa básico que nos permita conectar con el RDBMS y modificar algún dato.

EJEMPLO 9.3 `CSUpdate.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSUpdate`. Deberás ajustar el componente `TFDConnection` para conectar con el servidor SQL Server y abrir la base de datos `AdventureWorks`.

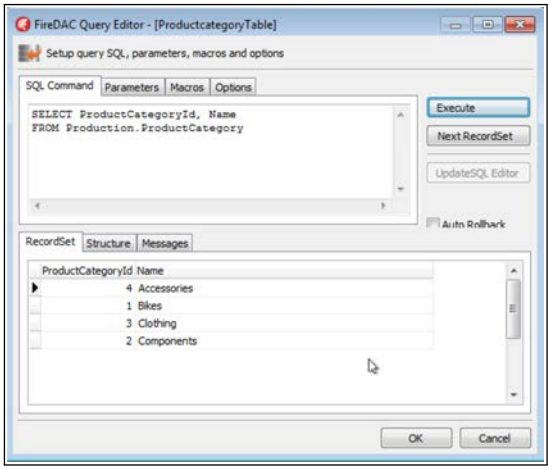


Figura 9.8 AJUSTAMOS LA CONSULTA A EJECUTAR

Partiendo de un proyecto vacío de tipo multi-dispositivo, al que añadimos un módulo de datos, tomamos la tabla `ProductCategory` desde el Explorador

de datos y la arrastramos hasta el módulo, configurando así la conexión y el `TFDQuery` asociado. A continuación editamos la consulta de dicho componente para seleccionar únicamente las columnas `ProductCategoryId`, que es la clave primaria, y `Name`, que contiene el nombre de la categoría (véase la Figura 9.8). En principio no modificaremos la propiedad `UpdateMode`. También agregaremos al módulo de datos un componente `TFDMoniRemoteClientLink` y, como es habitual, el `TFDGUIxWaitCursor`.

> **NOTA**
>
> *No olvides dar el valor `True` a la propiedad `Tracing` del `TFDMoniRemoteClientLink` y modificar el campo `MonitorBy` de la propiedad `Params` del `TFDConnection`, asignándole el valor `mbRemote`.*

En el formulario del proyecto introduciremos únicamente un componente `TGrid`, vinculándolo con el `TFDQuery`. Este control nos permitirá no sólo navegar por los datos, sino también introducir cambios en los campos mostrados. En este caso podríamos cambiar el nombre de las categorías.

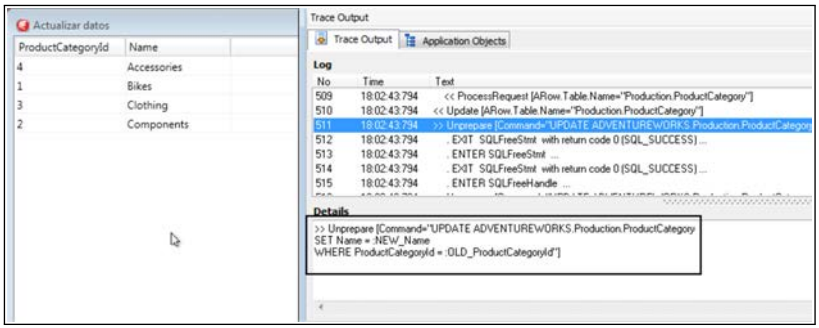


Figura 9.9 SE LOCALIZA LA FILA A ACTUALIZAR USANDO LA CLAVE PRIMARIA

Con la opción **TOOLS—FIREDAC MONITOR** abrimos el monitor de comunicación. A continuación ejecutamos el programa y modificamos el nombre de una de las categorías. En la traza de eventos apreciaremos, como se muestra en la Figura 9.9, que la sentencia `UPDATE` de actualización ha filtrado las

filas a modificar usando exclusivamente la clave primaria. En este contexto `OLD_ProductCategoryID` representa el valor para dicha columna que fue obtenido al recuperar los datos desde el servidor.

Si modificamos la propiedad `UpdateMode` del `TFDQuery`, asignándole el valor `upWhereChanged`, y repetimos la operación anterior comprobaremos cómo la sentencia de actualización ahora no solo utiliza la clave primaria para encontrar la fila afectada, sino que también verifica que los campos que se han cambiado, en este caso `Name`, tengan en la base de datos el valor que obtuvimos inicialmente (véase la Figura 9.10).

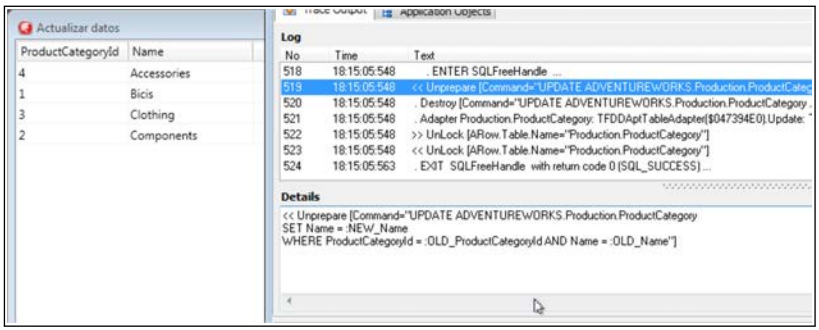


Figura 9.10 LA CLÁUSULA `WHERE` USA LA CLAVE PRIMARIA Y LOS CAMPOS MODIFICADOS

Anteriormente se apuntaba que, en caso de que la consulta introducida en el `TFDQuery` no incluyese entre sus columnas la clave primaria, la sentencia de actualización podría ser inadecuada. Es algo fácilmente comprobable: modifica la consulta del programa anterior para seleccionar únicamente la columna `Name`. A continuación vuelve a ejecutar el programa e introduce algún cambio, examinando la sentencia de actualización generada.

Como se aprecia en la Figura 9.11, la cláusula `WHERE` está usando como filtro únicamente la columna modificada. Si hubiese varias filas con el mismo valor en dicha columna, algo perfectamente posible, la actualización afectaría a todas ellas. De ahí la importancia de incluir siempre la clave primaria en la consulta, aunque no la necesitemos en la interfaz de usuario.

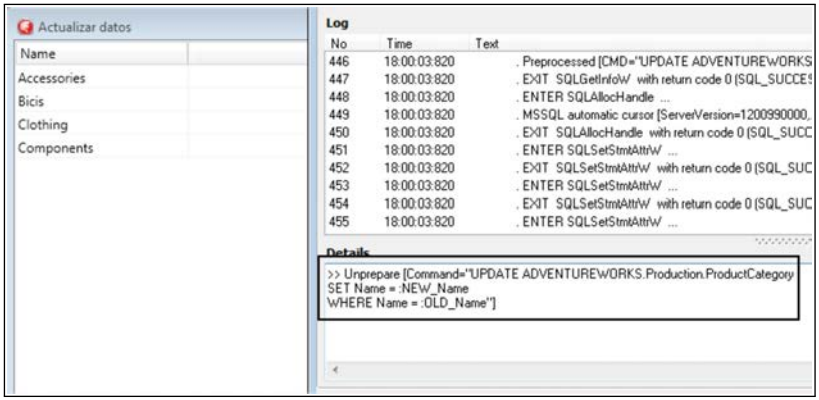


Figura 9.11 LA SENTENCIA DE ACTUALIZACIÓN PODRÍA AFECTAR A MÚLTIPLES FILAS

NOTA

Si el campo `CountUpdateRecords` *de la propiedad* `UpdateOptions` *del* `TFDQuery` *tiene el valor* `True`*, que es su valor por defecto, tras cada actualización se comprobará que el número de filas afectadas es exactamente 1, generándose una excepción en caso de no ser así y deshaciéndose la transacción de actualización previa. Por tanto, en la práctica no sería posible actualizar por accidente más filas de las deseadas, habría que dar el valor* `False` *a la citada propiedad* `CountUpdateRecords` *para permitirlo.*

Especificar la tabla a actualizar

En ocasiones la consulta efectuada no incluirá el nombre de la tabla que realmente hay que actualizar. Esto ocurre, por ejemplo, cuando se utiliza una vista como origen de la consulta o al componer consultas con campos agregados y relaciones, de forma que el resultado obtenido, en caso de que se modifique alguna fila del mismo, no puede volver a la entidad referenciada en la cláusula `FROM` de la sentencia `SELECT`.

Casos como los descritos en muchas ocasiones lo único que precisan conocer es la tabla sobre la que ha de aplicarse la actualización, al incluirse en la consulta todas las columnas necesarias para ello. El nombre de la tabla se facilitaría en este caso en el campo `UpdateTableName` de la propiedad `UpdateOptions`. Durante la fase de diseño no tenemos más que abrir la lista asociada a dicho campo y elegir la tabla de destino de la actualización.

Supongamos que tenemos un `TFDQuery` con una consulta como la mostrada a continuación, con la que queremos obtener el identificador y nombre de cada categoría y de las subcategorías que le corresponden, obteniendo los datos visibles en la Figura 9.12:

```
SELECT Category.ProductCategoryID, Category.Name,
       SubCategory.ProductSubCategoryID, SubCategory.Name
FROM Production.ProductSubcategory As SubCategory
INNER JOIN Production.ProductCategory As Category
          ON SubCategory.ProductCategoryID = Category.↙
              ↳ ProductCategoryID
```

Listado 9.6 Consulta no actualizable directamente

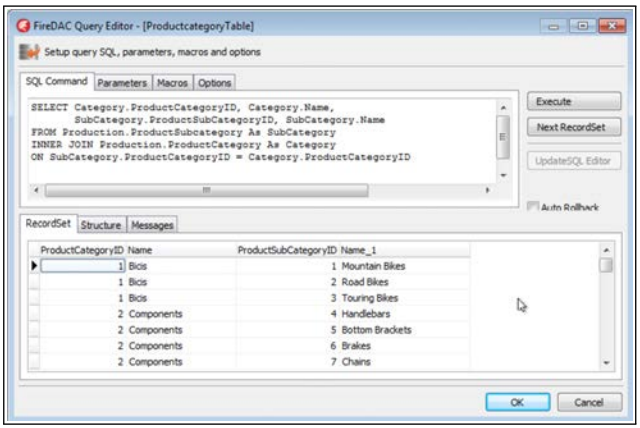


Figura 9.12 CONSULTA CON RELACIÓN ENTRE DOS TABLAS

Si conectamos este componente a un `TGrid` e intentamos modificar el nombre de una de las categorías[1], al invocar al método `Post` (lo cual ocurre automáticamente al cambiar de fila en la cuadrícula) se generará una excepción. La sentencia de actualización generada hace referencia a la tabla de subcategorías, ya que esta es la que aparece en la cláusula `FROM` de la consulta.

Para que la tabla a actualizar sea la correcta tendremos que seleccionar el `TFDQuery`, abrir la propiedad `UpdateOptions` en el Inspector de objetos, desplegar la lista asociada a la subpropiedad `UpdateTableName` y elegir la tabla `Production.ProductCategory`. A pesar de este ajuste, al intentar de nuevo modificar el nombre de una categoría nos encontraremos otra vez con una excepción. Usando la herramienta de monitorización, a fin de examinar la sentencia de actualización empleada (véase la Figura 9.13), descubriremos dónde está el problema: se está utilizando como clave en la actualización la clave primaria de la tabla de subcategorías, en lugar de la correspondiente a la tabla de categorías.

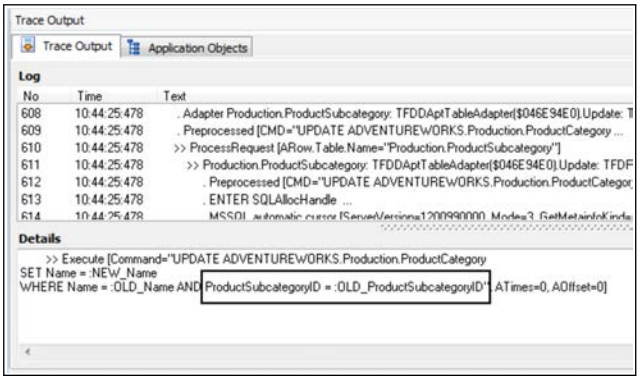


Figura 9.13 LA CLAVE PRIMARIA EMPLEADA NO ES LA CORRECTA

[1] Asumamos que el nombre de las subcategorías es de solamente lectura y no puede ser cambiado. Para que sea efectivamente así no tenemos más que abrir el Editor de campos asociado al `TFDQuery`, usar la opción **ADD ALL FIELDS** para añadir todos los campos de la consulta, seleccionar las columnas de la tabla de subcategorías y asignar el valor `True` a su propiedad `ReadOnly`.

Como se indicó con anterioridad, normalmente el componente `TFDQuery` es capaz de detectar automáticamente los campos que forman la clave primaria. Esto, no obstante, no es posible cuando se utilizan vistas o, como en este caso, una consulta con relaciones entre tablas. En estos casos recurriremos al campo `KeyFields` de la propiedad `UpdateOptions` para indicar explícitamente qué columnas forman la clave primaria. `KeyFields` tiene asociado un editor específico como el mostrado en la Figura 9.14. En la lista de la izquierda aparecen todas las columnas resultantes de la consulta, de los cuales agregaremos a la lista de la derecha aquellos que forman la clave primaria.

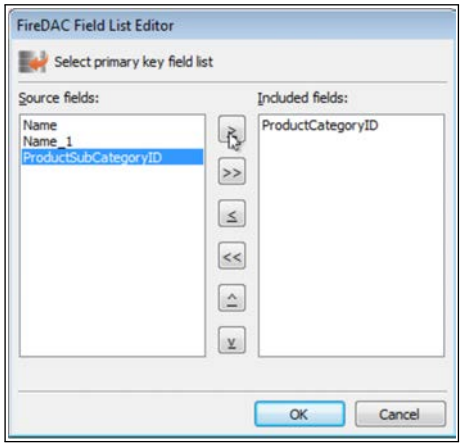


Figura 9.14 ESTABLECEMOS LA CLAVE PARA ACTUALIZAR

Tras realizar este último ajuste podemos volver a ejecutar la aplicación e intentar modificar el nombre de cualquier categoría. En esta ocasión la operación se ejecutará sin problemas, ya que la sentencia de actualización (véase la Figura 9.15) es totalmente correcta.

Personalización de las sentencias de actualización

A pesar de todas las posibilidades de ajuste que nos ofrecen las propiedades citadas en los apartados previos, en ocasiones la sentencia de actualización para un cierto conjunto de datos no podrá ser generada de forma automática. En estos casos hemos de facilitar una sentencia de actualización personalizada, escrita

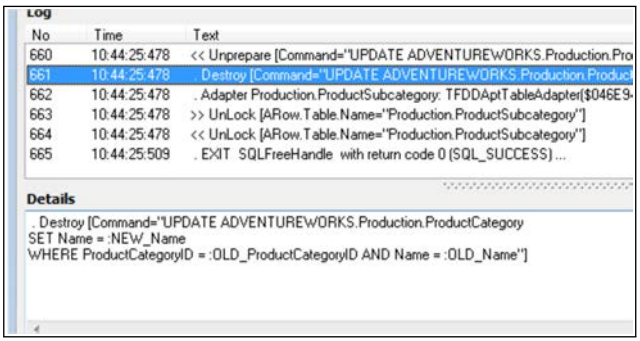


Figura 9.15 LA ACTUALIZACIÓN SE COMPLETA SIN PROBLEMAS

manualmente y ajustada a las necesidades concretas de cada caso. En realidad, en la mayoría de los casos tendremos que aportar tres sentencias SQL: una de actualización, otra de inserción y una tercera de borrado.

El componente necesario para introducir sentencias SQL personalizadas es `TFDUpdateSQL`. Este se conecta con el `TFDQuery` a través de la propiedad `UpdateObject` de este último. Las propiedades clave de `TFDUpdateSQL` son:

- `InsertSQL`: Contendrá la sentencia SQL a utilizar para introducir nuevas filas en la tabla. Los valores a insertar se referencian mediante la sintaxis `NEW_nombrecolumna`.

- `ModifySQL`: Esta propiedad aportará la sentencia SQL que se usará para actualizar datos existente en la tabla. Además de referencias a los nuevos valores, con la sintaxis antes indicada, normalmente también serán precisos valores recuperados con anterioridad, en la consulta previa a la actualización, para componer la cláusula `WHERE`. Dichos valores son accesibles mediante la sintaxis `OLD_nombrecolumna`.

- `DeleteSQL`: Almacenará la sentencia SQL encargada de eliminar filas de la tabla. Esta utilizará habitualmente la clave primaria en la sentencia `WHERE` a fin de afectar únicamente a la entrada que corresponda.

Además de estas tres, en ocasiones puede ser preciso utilizar también las propiedades `FetchRowSQL`, cuya sentencia facilitaría la recuperación de una fila concreta; `LockSQL`, con la sentencia a utilizar para bloquear una fila que va a ser modificada, y `UnlockSQL`, que facilitaría la sentencia SQL complementaria a la anterior.

> **NOTA**
>
> *El componente `TFDUpdateSQL` puede vincularse, además de a un `TFDQuery`, con los componentes `TFDTable` y `TFDStoredProc`. En este último caso es obligado utilizar un `TFDUpdateSQL`, ya que el componente `TFDStoredProc` no aporta sentencia de selección alguna que facilite la generación automática del resto de sentencias SQL.*

Las sentencias a almacenar en las anteriores propiedades pueden generarse durante la ejecución y simplemente asignarse, como cadenas de texto que son. Durante la fase de diseño, por el contrario, podemos recurrir al editor específico con que cuenta el componente `TFDUpdateSQL`. Veamos con un ejemplo cuáles serían los pasos a seguir para configurar dicho componente a fin de poder actualizar antes mostrada en el Listado 9.6. Partimos, como es habitual, de un proyecto con un módulo de datos, un componente `TFDConnection` conectado a la base de datos de ejemplo `AdventureWorks`, un `TFDGUIxWaitCursor` y un `TFDQuery`. A partir de aquí, el procedimiento a seguir sería el siguiente:

1. Agregamos al formulario un `TFDUpdateSQL`. Este se conectará automáticamente con el `TFDConnection`. Seleccionamos el `TFDQuery`, buscamos su propiedad `UpdateObject` y elegimos de la lista desplegable el único elemento disponible. De esta forma vinculamos el conjunto de datos con el componente que facilitará las sentencias SQL necesarias.
2. Hacemos doble clic sobre el `TFDQuery` para abrir el editor de consultas, introduciendo la consulta de selección propuesta en el Listado 9.6. Podemos ejecutarla para comprobar que el resultado es el esperado.
3. A continuación hacemos clic en el botón **UPDATESQL EDITOR** que hay en el editor de consultas (véase la Figura 9.16). Si el `TFDQuery` está correctamente vinculado al `TFDUpdateSQL` se abrirá el editor específico de este último.

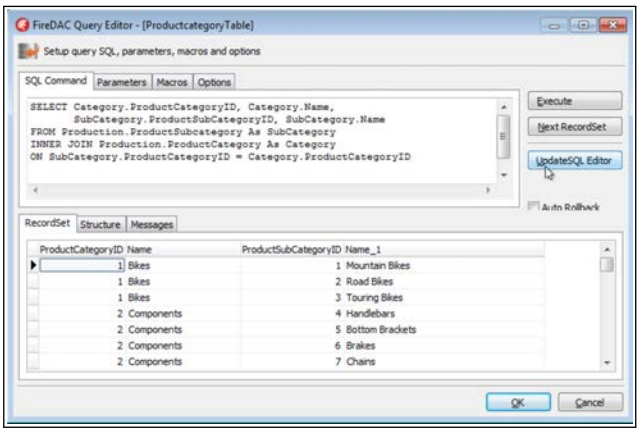


Figura 9.16 INTRODUCIMOS LA CONSULTA Y ACCEDEMOS AL EDITOR DEL COMPONENTE TFDUPDATESQL

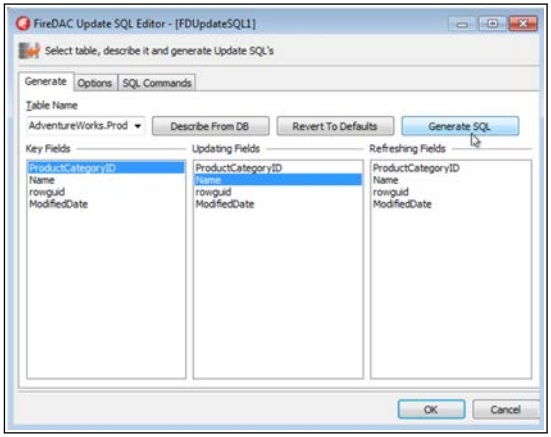


Figura 9.17 SELECCIONAMOS LAS COLUMNAS PARA CADA CATEGORÍA

4. La página **GENERATE** de este editor, mostrada en la Figura 9.17, nos permite generar automáticamente las sentencias SQL de inserción, modificación y eliminación. Para ello tenemos que dar los siguientes pasos:

(a) Abrimos la lista desplegable **TABLE NAME** y seleccionamos la tabla a actualizar. En este caso, asumiendo que tenemos la base de datos `AdventureWorks`, elegiríamos `Production.ProductCategory`.

(b) Seleccionamos en la lista **KEY FIELDS** las columnas que forman la clave primaria.

(c) Seleccionamos en la lista **UPDATING FIELDS** las columnas de la tabla que se podrán actualizar.

(d) En la última lista se pueden marcar los campos que es necesario actualizar tras las operaciones.

(e) Finalmente hacemos clic en el botón **GENERATE SQL**.

5. Para terminar, usando la página **SQL COMAMANDS** (véase la Figura 9.18) revisamos las sentencias SQL generadas y, si es necesario, introducimos cualquier ajuste que proceda. También podríamos utilizar esta página del editor para escribir manualmente dichas sentencias, en lugar de generarlas con los pasos previos.

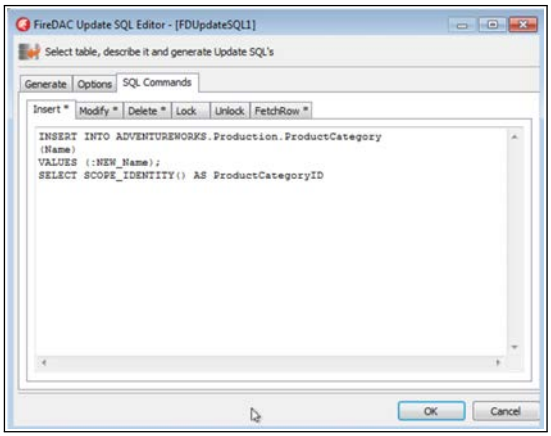


Figura 9.18 AJUSTAMOS LAS SENTENCIAS SQL SI ES PRECISO

6. Pulsamos el botón **OK** para cerrar el editor del componente `TFDUpdateSQL` y el mismo botón en el editor del componente `TFDQuery` cerrándolo también.

Para comprobar la configuración que hemos definido no tenemos más que agregar un `TGrid` al formulario, enlazarlo con el `TFDQuery`, dar el valor `True` a la propiedad `Active` de este y ejecutar el proyecto.

9.4 Procesamiento de datos en el servidor

En las secciones previas de este capítulo ya se ha mencionado la importancia que tiene en una configuración cliente/servidor la reducción del volumen de datos a transmitir. Una de las vías que tenemos para disminuir la cantidad de información a transferir consiste en trasladar, cuando sea posible y adecuado, el procesamiento de los datos desde el cliente al servidor.

Esta sección describe tres escenarios habituales en los que, trasladando el procesamiento al servidor, podemos incrementar de manera notable el rendimiento de nuestra aplicación en la mayoría de los casos. Lógicamente pueden darse multitud de excepciones. Si la comunicación entre el servidor de datos y los clientes es de alta capacidad y velocidad pero, por el contrario, el servidor de datos está saturado por la cantidad de solicitudes a procesar, las técnicas descritas a continuación serían contraproducentes.

9.4.1 Agrupación y agregado de datos

Siempre que necesitemos resultados agregados y/o agrupados de los datos que aloja una base de datos, en general será mucho más eficiente dejar el trabajo en manos del RDBMS que trasladar todas las filas de datos hasta el cliente y efectuar los cálculos en este.

El método posiblemente más eficiente para conseguir datos agrupados y agregado a partir de una consulta sea creando una vista en el RDBMS. Esta se almacenará ya analizada y compilada en el servidor de datos, pudiendo consultarse como si de una tabla real se tratase. No obstante, si no tenemos privilegios para crear este tipo de objeto en la base de datos o el RDBMS no los soporta, siempre podemos recurrir a enviar directamente la consulta incluyendo las funciones de agregado y agrupación apropiadas.

Supongamos que necesitásemos obtener el precio medio para cada subcategoría de producto existente en nuestra base de datos `AdventureWorks`. Para obtener esta información tenemos básicamente dos alternativas:

- Ejecutar una consulta sobre la tabla de productos obteniendo el código de subcategoría y precio de todas sus filas. Una vez recibidos los datos en el cliente, este se encargaría de recorrer dichas filas y efectuar el cálculo.

- Ejecutar una consulta en la que se soliciten los datos ya agrupados y agregados, dejando que sea el RDBMS quien realice el trabajo y nos devuelva los resultados.

Esta segunda opción es la que se pone en práctica en la Figura 9.19. En lugar de cientos o miles de filas, tantas como productos existan en la base de datos, se recibe solo una fila por subcategoría incluyendo ya el precio medio. El volumen de datos transferidos por la red, así como el procesamiento efectuado en el cliente, se reducen drásticamente.

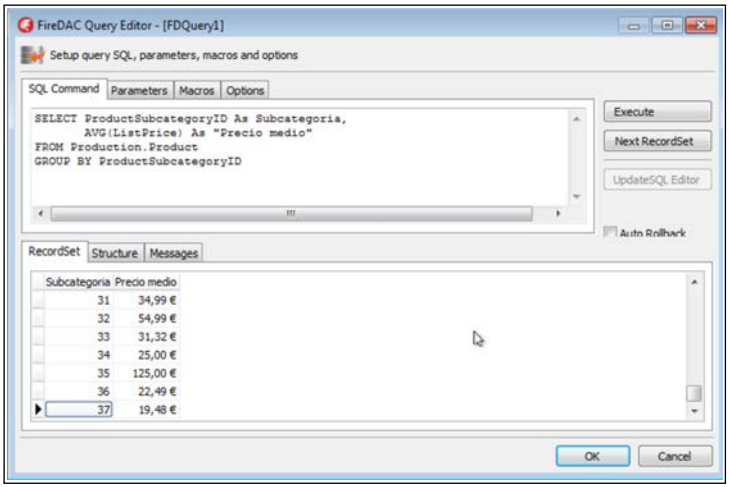


Figura 9.19 SOLICITAMOS AL SERVIDOR LOS DATOS AGREGADOS Y AGRUPADOS

9.4.2 Otros cálculos sobre los datos

Cuando los cálculos a efectuar sobre los datos no se limitan a sumas, conteos y promedios, que es lo que nos ofrecen las funciones de agregado de SQL, puede parecer que la única alternativa consiste en trasladar toda la información hasta el cliente para que este pueda elaborar los resultados que se necesitan. Existe, no obstante, una alternativa: la creación en el RDBMS de las funciones apropiadas para efectuar dicho trabajo.

La mayor parte de los RDBMS contemplan la definición de funciones que, aceptando un conjunto de parámetros variable como entrada, pueden acceder al contenido de las tablas, ya sea mediante consultas o con cursores, y realizar prácticamente cualquier cálculo sobre los datos y devolver un resultado. Este puede ser un valor escalar, por ejemplo un número, o bien un conjunto de filas y columnas, según los casos.

Imaginemos que queremos trasladar a una función la obtención de los códigos de subcategoría de productos con sus respectivos precios medios, el mismo resultado que obteníamos anteriormente con una consulta, pero limitando las filas obtenidas a aquellos casos en los que el precio medio supere un cierto límite. El supuesto podría ser más complejo e implicar el uso de construcciones más elaboradas del lenguaje T-SQL de SQL Server, pero aquí nos centramos específicamente en cómo utilizar la función desde Delphi. El primer paso sería abrir la base de datos `AdventureWorks` y ejecutar el siguiente código para crear la función:

```
1  CREATE FUNCTION HighPriceCategories (@LimitPrice money)
2  RETURNS @HighPricesTable TABLE (
3      SubCategoryId     int,
4      AveragePrice      money
5  )
6  AS
7  BEGIN
8     INSERT @HighPricesTable
9          SELECT S.ProductSubcategoryID,
10                AVG(P.ListPrice) As AveragePrice
11         FROM Production.ProductSubcategory AS S
12         INNER JOIN Production.Product AS P
```

```
13              ON S.ProductSubcategoryID = P.↙
                    ↳ ProductSubcategoryID
14          GROUP BY S.ProductSubcategoryID
15          HAVING AVG(P.ListPrice) > @LimitPrice
16    RETURN
17 END
```

Listado 9.7 Función para SQL Server

EJEMPLO 9.4 `CSFuncionSQL.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSFuncionSQL`. Deberás ajustar el componente `TFDConnection` para conectar con el servidor SQL Server y abrir la base de datos `AdventureWorks`. Previamente has de crear la función mostrada en el Listado 9.7 en dicha base de datos, ya sea desde el Explorador de datos de Delphi o recurriendo al SQL Server Management Studio.

Creada la función, iniciaremos un nuevo proyecto de tipo multi-dispositivo al que agregaremos un módulo de datos. En este insertaremos los componentes necesarios para conectar con la base de datos, incluyendo un `TFDQuery`. Hacemos doble clic sobre él e introducimos la consulta a ejecutar, que será la siguiente:

```
1 SELECT *
2 FROM HighPriceCategories(:LimitPrice)
```

Listado 9.8 Consulta para ejecutar la función SQL

En esta consulta se ha introducido un parámetro sustituible que es necesario configurar. Para ello recurrimos a la página **PARAMETERS** del mismo cuadro de diálogo (véase la Figura 9.20), indicando que se trata de un parámetro de entrada hacia la función de tipo moneda y facilitando un valor por defecto. Podemos usar el botón **EXECUTE** para comprobar que la consulta funciona correctamente.

La interfaz de usuario del programa contará con un `TPanel` ajustado a la parte superior y un `TGRid` ocupando el resto del espacio disponible. Enlazare-

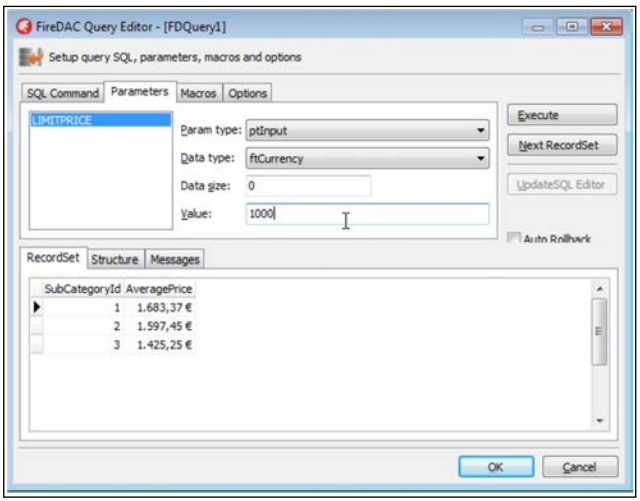


Figura 9.20 CONFIGURAMOS EL PARÁMETRO A FACILITAR A LA CONSULTA

mos este último con el `TFDQuery`. En el `TPanel` insertaremos un `TEdit`, que permitirá introducir el valor límite, y un botón que se encargará de actualizar los datos mostrados en la cuadrícula. Hacemos doble clic sobre el botón e introducimos el código mostrado a continuación:

```
1 procedure TfrmMain.Button1Click(Sender: TObject);
2 begin
3   with dmFuncionSQL.FDQuery1 do
4   begin
5     Disconnect;
6     Params.ParamByName('LimitPrice').Value  := StrToInt(↙
        ↳ Edit1.Text);
7     Open;
8   end;
9 end;
```

Listado 9.9 Código asociado al botón de consulta

Con esto ya estamos en disposición de ejecutar el proyecto y, como se aprecia en la Figura 9.21, introducir cualquier valor y ejecutar la consulta que invoca a la función antes definida. El trabajo de selección y filtrado de los datos se realiza completamente en el servidor, transfiriéndose al cliente solo unas pocas filas.

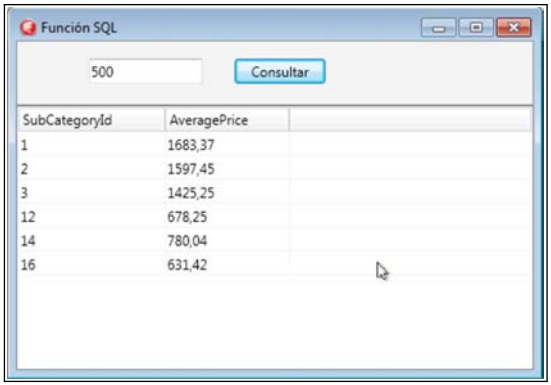


Figura 9.21 EL PROGRAMA EJECUTANDO LA FUNCIÓN

9.4.3 Actualización de datos por lotes

La mayor parte de los programadores tienden a confiar en el código de su aplicación para llevar a cabo todas las tareas de inserción y actualización de datos. Un administrador de bases de datos, por el contrario, centralizaría una buena parte de la lógica de tratamiento de los datos, especialmente aquellas operaciones repetitivas y de cierta complejidad, en el proprio servidor de datos.

En aquellos casos en los que es necesario actualizar lotes de datos, aplicando la misma operación a cada fila afectada, podemos mejorar la experiencia del usuario si en lugar de transferir los datos hasta el cliente, modificarlos en este y enviarlos de vuelta al servidor, efectuamos el trabajo directamente en este último. Para ello lo habitual es recurrir a la ejecución de procedimientos almacenados, bloques de código similares a las funciones pero que no han de devolver necesariamente un resultado.

> **NOTA**
>
> *La cláusula `SET` de una sentencia SQL `UPDATE` puede incluir cálculos y referencias a otras columnas, por lo que operaciones de actualización por lotes básicas, como puede ser la modificación de una columna en un cierto porcentaje o la asignación de un valor obtenido a partir de cálculos sobre otras columnas, no precisan el uso de procedimientos almacenados.*

Para ejecutar un procedimiento almacenado recurriremos habitualmente al componente `TFDStoredProc`. El nombre del procedimiento almacenado se facilitará en la propiedad `StoredProcName`. En el caso concreto de SQL Server, las propiedades `CatalogName` y `SchemaName` aportarán el nombre de la base de datos y el esquema, datos que pueden introducirse directamente en el nombre como prefijos separados por puntos. Para ejecutar el procedimiento `uspGetManagerEmployees` del esquema `dbo` de la base de datos `AdventureWorks`, por ejemplo, podríamos usar las tres propiedades o bien asignar el valor `AdventureWorks.dbo.uspGetManagerEmployees` a la propiedad `StoredProcName`.

La mayoría de las veces el procedimiento almacenado a ejecutar necesitará algún parámetro de entrada. Estos se facilitarán, como es habitual, mediante la colección `Params` del componente. Es una operación que podemos efectuar en la fase de diseño, mediante el editor asociado, o bien en ejecución, como se mostraba en el ejemplo previo al ejecutar una función.

9.5 Ejecución de scripts SQL

A pesar de que el componente `TFDConnection` nos permite, a través del método `ExecSQL`, ejecutar sentencias SQL, algo que también podemos hacer de forma general mediante el componente `TFDQuery`, en ocasiones podemos encontrarnos con ciertas limitaciones. Determinados RDBMS no permiten usar por esta vía algunos tipos de sentencias, como las de definición de estructuras de datos. Tampoco se suele permitir la declaración de variables, uso de cursores y otras técnicas habitualmente reservadas para los *scripts* SQL.

Para poder ejecutar un *script* SQL desde una aplicación Delphi lo más cómodo, salvo que tengamos el código del *script* implementado en el RDBMS como un procedimiento o función que podamos ejecutar directamente, es recurrir al componente `TFDScript`. Este ofrece una vía para ejecutar *scripts* SQL bajo el control de nuestro programa.

En esta sección se explica, y también se demuestra mediante un ejercicio simple, cómo utilizar el componente `TFDScript` para ejecutar guiones SQL desde nuestra aplicación.

9.5.1 Vías para aportar el código SQL a ejecutar

Las sentencias SQL que componen el guión a ejecutar pueden entregarse al componente `TFDScript` por distintas vías, de las cuales elegiremos la que más nos convenga en cada caso. Dichas alternativas son:

- **En un archivo**: Es habitual almacenar los guiones SQL, especialmente cuando son relativamente extensos, en un archivo independiente. Este suele tener extensión `.sql`. Podemos asignar la ruta y nombre del archivo a la propiedad `SQLScriptFileName` del componente, distribuyendo el guión junto con la aplicación. Esto también nos permitiría alterar el código SQL sin necesidad de modificar el proyecto.

- **En una colección de cadenas**: La propiedad `SQLScripts` del componente `TFDScript` puede contener una colección de objetos, cuya clase es `TFDSQLScript`, compuesto cada uno de ellos de un nombre y una colección de cadenas de texto (propiedad `SQL`) pensadas para almacenar las sentencias del *script*. Esta propiedad puede ser editada tanto en la fase de diseño (véase el siguiente apartado) como en ejecución.

- **Como parámetro**: Mediante el método `ExecuteScript` es posible ejecutar un *script* facilitándolo directamente como parámetro. El tipo del argumento de entrada es `TStrings`, una colección de cadenas de texto.

Como se aprecia en la Figura 9.22, además de las dos propiedades ya citadas el componente `TFDScript` cuenta también con otras que ya conocemos, como `FetchOptions` o `Params`, a las que se suman algunas más de carácter específico, como es el caso de `ScriptOptions`. Esta última es un objeto de

clase `TFDScriptOptions` conteniendo una extensa lista de propiedades que determinan la forma en que se ejecutará el *script*. Por regla general los valores por defecto serán los adecuados.

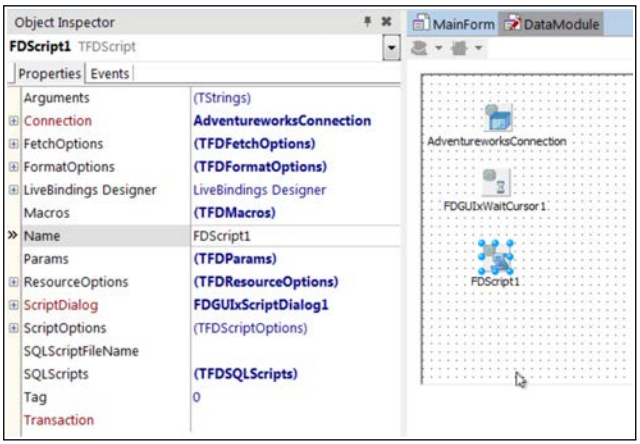


Figura 9.22 PROPIEDADES DEL COMPONENTE `TFDSCRIPT`

> **NOTA**
>
> *El componente `TFDScript` reconoce la sintaxis para scripts SQL de los RDBMS más populares, incluyendo SQL Server, Oracle, Interbase, Firebird y MySQL.*

9.5.2 Edición y prueba del guión en la fase de diseño

Mientras nos encontramos en el diseñador de Delphi podemos probar los guiones SQL que pretendemos utilizar usando las dos primeras vías de las tres antes citadas. En la primera bastaría con facilitar el nombre del archivo con el guión. En la segunda tendremos que abrir el editor específico asociado a la propiedad `SQLScripts`, haciendo a continuación clic en el primer botón para añadir al

menos un elemento. Este se abrirá de inmediato en el Inspector de objetos, en el que veremos que cuenta con dos propiedades: `Name` y `SQL`. Esta última tiene, a su vez, un editor específico que será el que utilicemos para introducir el guión, tal y como se muestra en la Figura 9.23.

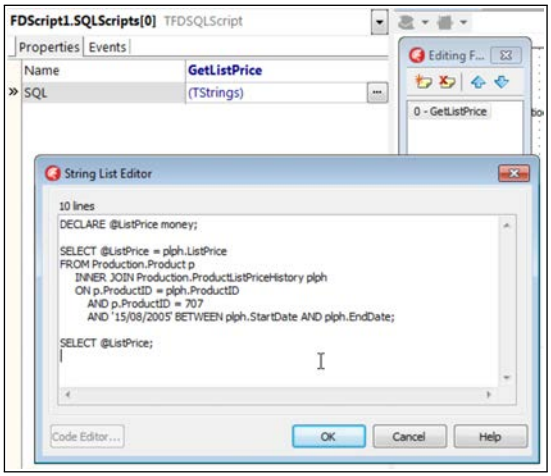


Figura 9.23 EDICIÓN DEL *script* SQL

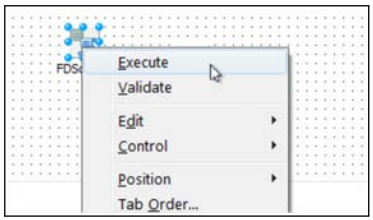


Figura 9.24 OPCIONES PARA PROBAR EL GUIÓN EN LA FASE DE DISEÑO

Una vez que hemos introducido el guión, o indicado la ruta y nombre del archivo, las opciones **VALIDATE** y **EXECUTE** del menú contextual (véase la Figura 9.24) del componente `TFDScript` nos permiten validarlo y ejecutarlo, respectivamente. Habitualmente, salvo que el guión genere algún error, solamente veremos aparecer momentáneamente un cuadro de diálogo, sin más. Para

interactuar con el guión tendremos que responder a eventos de `TFDScript` como `OnConsoleGet` y `OnConsolePut`. Como alternativa podemos recurrir al componente `TFDGUIxScriptDialog` descrito en el siguiente apartado.

9.5.3 Asociar un cuadro de diálogo específico

El componente `TFDScript` cuenta con los eventos necesarios para facilitar la interacción con el guión a medida que se ejecuta. La validación y ejecución se lleva a cabo mediante los métodos `ValidateStep`, `ValidateAll`, `ExecuteStep` y `ExecuteAll`. Con el primero se validan paso a paso las partes del guión, mientras que con el segundo se valida por completo. Los otros dos ejecutan un paso o todo el guión, respectivamente. La posición actual en el guión, útil cuando se ejecuta paso a paso, viene dada por la propiedad `Position`.

Si queremos aportar una interfaz gráfica que permita al usuario interactuar con el guión, introduciendo valores si fuesen necesarios y obteniendo tanto información sobre los errores como la visualización de los resultados, podemos usar el componente `TFDGUIxScriptDialog`. Al igual que ocurre con `TFDGUIxWaitCursor`, el citado componente cuenta con una propiedad llamada `Provider` que determina si la interfaz será FMX, VCL o de consola.

Tras agregarlo al módulo de datos o formulario, el `TFDGUIxScriptDialog` se vincula al `TFDScript` mediante la propiedad `ScriptDialog` de este último. Con la propiedad `Options` del primero es posible ajustar algunas características del funcionamiento del cuadro de diálogo.

9.5.4 En la práctica

Comprobemos con un ejercicio cuáles serían los pasos a seguir para introducir y ejecutar un *script* SQL desde un programa. Iniciamos un proyecto de tipo multi-dispositivo, agregamos un módulo de datos e introducimos en el un componente `TFDConnection` configurándolo para acceder a la base de datos SQL Server `AdventureWorks`. Incluimos también un componente `TFDGUIxWaitCursor`, un `TFDScript` y un `TFDGUIxScriptDialog`. Todos ellos se enlazarán automáticamente con el `TFDConnection`. A continuación seguimos estos pasos:

1. Abrimos el editor de la propiedad `SQLScripts` del `TFDScript`, hacemos clic en el primer botón para crear un guión y le asignamos el nombre `GetListPrice`.

2. Abrimos el editor asociado a la propiedad `SQL` del elemento creado en el paso anterior, introduciendo el guión SQL mostrado a continuación. Este tiene por objetivo recuperar el precio que tenía un determinado producto en una fecha dada, consultando para ello una tabla en la que están registrados los precios históricos de todos los productos.

```
1  DECLARE @ListPrice money;
2
3  SELECT @ListPrice = plph.ListPrice
4  FROM Production.Product p
5       INNER JOIN Production.ProductListPriceHistory plph
6       ON p.ProductID = plph.ProductID
7           AND p.ProductID = 707
8           AND '15/08/2005' BETWEEN plph.StartDate AND plph↙
                ↳ .EndDate;
9
10 SELECT @ListPrice;
```

Listado 9.10 Guión SQL para obtener el precio

3. Vinculamos el `TFDScript` con el `TFDGUIxScriptDialog`. Para ello usamos la lista asociada a la propiedad `ScriptDialog` del primero. La abrimos y elegimos el único elemento existente.

4. Desplegamos los campos de la propiedad `Options` correspondiente al componente `TFDGUIxScriptDialog` y desactivamos `ssAutoHide`. Con esto evitamos que se oculte automáticamente el cuadro de di´alogo una vez que la ejecución del guión haya finalizado.

5. Cambiamos al formulario, que enlazaremos con el módulo de datos, y agregamos al mismo un `TButton`. Este será el único elemento de nuestra interfaz de usuario.

6. Hacemos doble clic sobre el botón e introducimos las dos sentencias siguientes a fin de validar y ejecutar el guión:

```
1 procedure TfrmMain.Button1Click(Sender: TObject);
2 begin
3   dmScript.FDScript1.ValidateAll;
4   dmScript.FDScript1.ExecuteAll;
5 end;
```

Listado 9.11 El botón validará y ejecutará el guión SQL

Ahora ya podemos ejecutar el proyecto. Al hacer clic sobre el botón que habíamos introducido en el formulario debe abrirse automáticamente el cuadro de diálogo mostrado en la Figura 9.25, facilitando en la parte inferior el resultado que ha generado la ejecución del guión SQL. Si se produjese algún error también aparecería toda la información relativa al mismo en el panel.

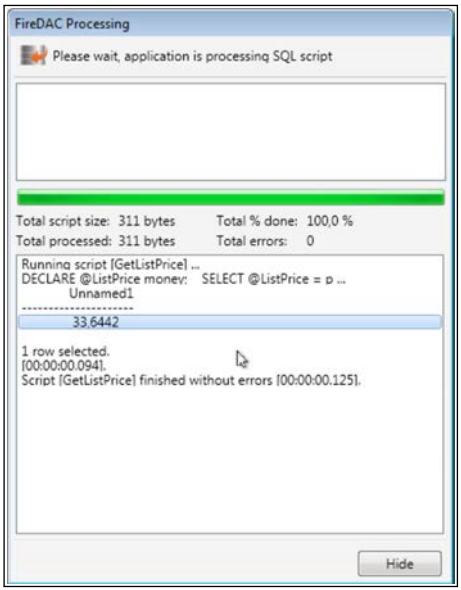


Figura 9.25 CUADRO DE DIÁLOGO CON LA EJECUCIÓN DEL SCRIPT

EJEMPLO 9.5 `CSScript.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSScript`. Deberás ajustar el componente `TFDConnection` para conectar con el servidor SQL Server y abrir la base de datos `AdventureWorks`.

9.6 Resumen

CT V-Julio García, a lo largo de este capítulo hemos conocido multitud de aspectos que es necesario tomar en consideración a la hora de desarrollar una aplicación cliente/servidor con Delphi. Al conectar con el RDBMS es necesario autenticarse de manera apropiada, así como optimizar en la medida de lo posible el uso que se hace de las conexiones con el servidor de datos. También se han descrito los procedimientos a seguir para obtener y editar datos en este contexto, así como las vías que nos permiten realizar parte del trabajo en el propio servidor, reduciendo la cantidad de información que es necesario transportar por la red hasta los dispositivos que actúan como clientes. Finalmente se ha explicado cómo ejecutar guiones SQL directamente desde Delphi, sin necesidad de escribir funciones o procedimientos almacenados ni recurrir a las consolas interactivas ofrecidas por cada RDBMS.

En el siguiente capítulo abordaremos un tema que no nos ha preocupado hasta el momento: la gestión de transacciones. En un entorno mono-usuario, como el de los proyectos propuestos en la primera parte del libro, este es un aspecto que casi carece de importancia. En un contexto cliente/servidor, por el contrario, resulta vital.

Capítulo 10

TRANSACCIONES, BLOQUEOS Y NOTIFICACIÓN DE CAMBIOS

Cuando una aplicación manipula la información alojada en un servidor de datos, usando para ello sentencias SQL a través de los componentes descritos en capítulos previos, siempre lo hace en el contexto de una transacción. FireDAC cuenta con un mecanismo de gestión de transacciones por defecto, razón por la que, hasta ahora, no nos hemos preocupado de este aspecto.

El uso de transacciones es especialmente importante cuando se trabaja con datos procedentes de dos o más tablas o incluso con información procedente de bases de datos distintas. El ejemplo más típico de la importancia de las transacciones es el relativo a los movimientos bancarios: un usuario ordena una trans-

ferencia desde una cuenta a otra, lo cual implica realizar dos anotaciones (dos modificaciones sobre la base de datos), la primera con el cargo en la cuenta de origen y la segunda con el abono en la cuenta de destino. Si se produjese algún tipo de fallo entre ambas operaciones el estado de la base de datos sería inconsistente.

El objetivo de este capítulo es exponer la utilidad de las transacciones y la forma en que se gestionan al trabajar con bases de datos mediante los componentes de FireDAC. Comenzaremos conociendo las propiedades que aporta una transacción, para a continuación distinguir entre el control de transacciones ofrecido por el RDBMS, accesible desde los guiones SQL, y el propio de FireDAC, de nivel más genérico.

10.1 Introducción al uso de transacciones

En las ocasiones en que **una operación lógica** sobre los datos implica la ejecución de **múltiples operaciones físicas** sobre la información almacenada en la base de datos, hay multitud de situaciones de este tipo similares a la mencionada antes relativa a la transacción bancaria, el uso de transacciones resulta indispensable.

Esta sección introduce los fundamentos relativos al uso de transacciones en un RDBMS, enumerando las propiedades que aportan y las operaciones básicas de control.

10.1.1 Propiedades de una transacción

El correcto uso de transacciones durante el desarrollo de una aplicación cliente/servidor aportará las siguientes propiedades:

- **Atomicidad**: La atomicidad de una transacción garantiza que la operación logica sea tratada como una sola operación física, de forma que se completen todos sus pasos o ninguno de ellos. No existe la posibilidad de que la transacción quede a medias.

- **Consistencia**: Esta propiedad garantiza el cumplimiento de todas las reglas aplicables a las operaciones que sean ejecutadas en el contexto de la transacción. Estas pueden ser restricciones de dominio, ejecución de desencadenadores, etc.

- **Aislamiento**: Las operaciones intermedias efectuadas en una transacción no son visibles para el resto de las transacciones, de forma que cada una se ejecuta de manera aislada respecto a las demás. Únicamente cuando una transacción se completa los cambios efectuados sobre los datos son visibles para el resto.

- **Durabilidad**: La durabilidad de una transacción es la propiedad que garantiza que, una vez se haya concluido y confirmado, los cambios efectuados en los datos serán perdurables incluso si se produjese cualquier fallo posterior que pudiera afectar a la base de datos. Para ello cada RDBMS suele contar con un mecanismo propio que, paralelamente a la propia base de datos, mantiene un registro de las operaciones aplicadas.

NOTA

Las propiedades que acaban de enumerase suelen conocerse por el acrónimo ACID, formado por la inicial de cada propiedad en inglés: Atomicity, Consistency, Isolation y Durability.

10.1.2 Operaciones de control de una transacción

Todos los RDBMS que contemplan el uso de transacciones, entre los que se incluyen todos los que soportan conexiones remotas y, por tanto, pueden ser usados en una configuración cliente/servidor, ofrecen un mecanismo de control de transacciones basado en las tres siguientes funciones:

- **Inicio de la transacción**: Marca el punto a partir del cual las sentencias SQL serán consideradas parte de una transacción. Dependiendo del RDBMS esta operación se efectuará con la orden `START TRANSACTION`, `BEGIN TRANSACTION` o similar.

- **Confirmación de la transacción**: Tras facilitar la secuencia de sentencias SQL que han de ejecutarse dentro de la transacción, esta se confirma para completar la ejecución. La sentencia más habitual en los RDBMS para esta operación es `COMMIT`.

- **Cancelación de la transacción**: Si durante la ejecución de las sentencias SQL que forman la transacción se produce algún fallo, la transacción completa es cancelada de forma que ninguna de las operaciones se aplica finalmente. La sentencia más habitual para esta operación es `ROLLBACK`.

Dependiendo de cada RDBMS las transacciones podrán anidarse o no, se contemplará el uso de transacciones continuas, etc.

10.2 Transacciones de base de datos vs transacciones FireDAC

Al desarrollar una aplicación cliente/servidor con Delphi y FireDAC tendremos fundamentalmente dos alternativas a la hora de usar transacciones: recurrir a las sentencias del RDBMS que vaya a utilizarse o bien confiar en el mecanismo ofrecido por FireDAC para esa misma tarea. En realidad estas dos opciones no son excluyentes entre sí.

Para utilizar el mecanismo de control de transacciones del RDBMS lógicamente debemos conocer las sentencias específicas que este ofrece, ya que estas varían según el producto. En contraposición, FireDAC nos ofrece una interfaz unificada para gestionar las transacciones. Será el componente `TFDConnection` el que se encargue, según el tipo de RDBMS al que esté conectado, de traducir nuestras acciones al conjunto de sentencias adecuado.

En general podemos encontrarnos con dos situaciones diferentes en las que utilizaríamos las transacciones de la base de datos o las de FireDAC:

- Al ejecutar *scripts* SQL, así como el interior de funciones, desencadenadores y procedimientos almacenados, recurriremos al mecanismo de control de transacciones ofrecido por el RDBMS.

- Al operar sobre los datos con componentes como `TFDQuery` o `TFDTable` el método preferente de control de transacciones será el que nos ofrece FireDAC.

En la siguiente sección centramos nuestra atención sobre el procedimiento de control de transacciones propio de FireDAC que, como programadores, será el que habitualmente usemos con mayor frecuencia.

10.3 Control de transacciones en FireDAC

FireDAC nos ofrece un mecanismo de control de transacciones de alto nivel, capaz de operar de forma autónoma como hemos podido comprobar en los ejercicios del capítulo previo, en el que en ningún caso hemos iniciado, confirmado o cancelado una transacción. Este sistema automático de control de transacciones, no obstante, también tiene sus limitaciones, de ahí que exista la posibilidad de aplicar un control manual sobre el estado de las mismas.

En esta sección se explican las diferentes alternativas que tenemos a nuestro alcance, así como los componentes, propiedades y métodos implicados en cada una de ellas.

10.3.1 Transacciones automáticas

El modo de trabajo de FireDAC consiste en iniciar automáticamente una transacción antes de ejecutar cualquier sentencia de actualización de datos (entiéndase inserción, modificación o eliminación), confirmándola o cancelándola dependiendo de que la sentencia SQL pueda ser finalmente ejecutada de forma satisfactoria o no.

El comportamiento de las transacciones automáticas viene determinado por los valores asignados a los distintos campos de la propiedad `TxOptions` del componente `TFDConnection` (véase la Figura 10.1). Como puede apreciarse, la mayor parte de ellas son booleanas.

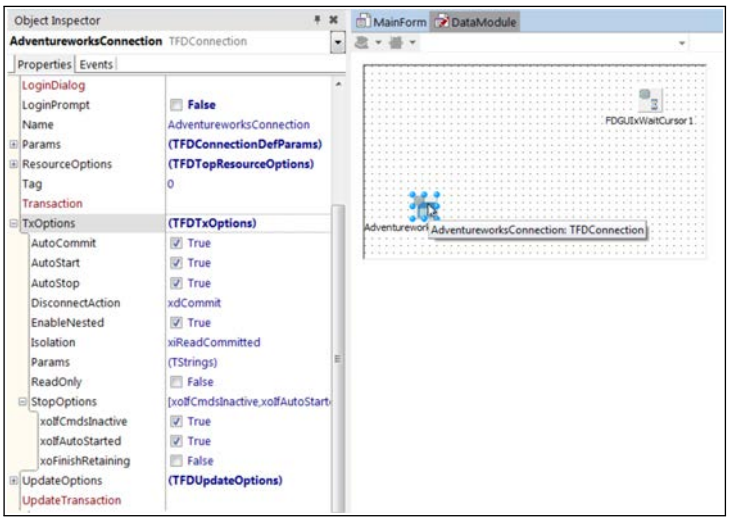


Figura 10.1 PROPIEDADES QUE CONTROLAN LA GESTIÓN DE TRANSACCIONES

Por defecto la propiedad `AutoCommit` siempre tendrá el valor `True`, activando así el mecanismo de gestión automático de transacciones. La mayor parte de RDBMS contempla este modo automático, pero hay excepciones como es el caso de InterBase. En estos casos FireDAC puede simular ese funcionamiento automático mediante las propiedades `AutoStart`, `AutoStop` y `StopOptions`. Las dos primeras indican si ha de iniciarse y concluirse cada transacción de forma automática, mientras que la última determina en qué caso se finalizará la transacción. Lo habitual es que activemos las opciones `xoIfAutoStarted` y `xoIfCmdsInactive`, de forma que la transacción solamente se finalice si fue iniciada automáticamente, lo que implica que la propiedad `AutoStart` tenga el valor `True`, y siempre esperando a que todos los comandos asociados a la conexión estén inactivos, es decir, no haya ninguna operación en curso.

Las transacciones automáticas son cómodas, ya que no tenemos que preocuparnos de su control, pero a cambio tienen algunos inconvenientes. Los dos fundamentales son la peor eficiencia del servidor de datos, sobre todo cuando se ejecutan muchas operaciones sobre él desde la aplicación y cada una de ellas se engloba en una transacción individual, y el hecho de que no permiten controlar

operaciones complejas que impliquen varias acciones sobre la base de datos. En estos casos es recomendable recurrir a las transacciones manuales.

10.3.2 Transacciones manuales

Aunque podemos dar el valor `False` a la propiedad `AutoCommit` para desactivar la gestión automática de transacciones, en realidad esto no es necesario ya que se desactivará por sí sola en cuanto iniciemos una nueva transacción de manera explícita, reactivándose cuando dicha transacción finalice. Los métodos que encontramos en el componente `TFDConnection` para control de transacciones son los siguientes:

- `StartTransaction`: Inicia una nueva transacción configurada según las opciones activas en la propiedad `TxOptions`. Las sentencias SQL de manipulación de datos enviadas a continuación se ejecutarán en el contexto de la nueva transacción.

- `Commit`: Este método hace permanentes los cambios efectuados sobre los datos, como resultado de la ejecución de sentencias DML, desde que se inició la transacción, habitualmente poniendo fin a esta.

- `Rollback`: Es el método complementario a `Commit`, encargándose de deshacer todos los cambios que se hayan efectuado sobre los datos y terminar la transacción.

> **NOTA**
>
> *Las transacciones únicamente afectan a sentencias SQL de tipo DML, las de manipulación de datos, no a las de tipo DDL, por lo que no tiene sentido usarlas a la hora de crear, modificar o eliminar estructuras de tablas, vistas y otros objetos de la base de datos.*

Al trabajar con transacciones manuales podemos consultar la propiedad booleana `InTransaction` del componente `TFDConnection` para saber si hay una transacción activa.

10.3.3 Configuración de las transacciones

Con independencia de que el control sea automático o manual, las transacciones de FireDAC utilizarán la configuración establecida por varios de los campos de la propiedad `TxOptions`. En general los valores por defecto que toman dichas subpropiedades resultarán adecuados en la mayoría de los casos. Los parámetros fundamentales vienen dados por:

- `DisconnectAction`: El valor asignado a esta propiedad determina qué acción se llevará a cabo en caso de que la conexión con la base de datos se cierre mientras hay activa una transacción. Su valor por defecto es `xdCommit`, indicando que la transacción se confirmará. Los otros valores posibles son `xdRollback`, para cancelar la transacción, y `xdNone`, para dejar en manos del RDBMS la acción a ejecutar.

- `ReadOnly`: Esta propiedad booleana sirve para indicar al RDBMS que durante la transacción no se efectuarán cambios a los datos. Su valor por defecto es `False` ya que, en la mayoría de los casos, una transacción se utiliza precisamente para garantizar las propiedades ACID sobre un conjunto de cambios.

- `Params`: Algunos RDBMS aceptan parámetros adicionales a la hora de trabajar con transacciones. La finalidad de esta propiedad es facilitar dichos parámetros. Se trata de un objeto `TStrings` que contendrá un parámetro por línea.

- `Isolation`: El objetivo de esta propiedad es fijar el nivel de aislamiento con el que se quiere trabajar en una transacción. Los valores que puede tomar son los cuatro indicados a continuación:

  - `xiReadCommitted`: Es el valor por defecto. Permite que la transacción en curso pueda leer cambios producidos por otras transacciones ya confirmadas.
  - `xiRepeatableRead`: Con esta configuración la transacción sólo efectuará una lectura de las filas sobre las que va a trabajar, por lo que no verá ningún cambio efectuado por otras transacciones simultáneas aunque se confirmen.

- `xiDirtyRead`: Es el nivel de menor aislamiento, al permitir que sean visibles para la transacción en curso cambios aún no confirmados efectuados en otras transacciones simultáneas.

- `xiUnspecified`: Con este valor se deja en manos del RDBMS la selección del modo de aislamiento.

Salvo que trabajemos con InterBase raramente tendremos que utilizar parámetros adicionales para las transacciones, por lo que la propiedad `Params` no nos será de mayor utilidad.

10.3.4 Transacciones anidadas

Uno de los campos booleanos de la propiedad `TxOptions` no mencionados hasta ahora es `EnableNested`. Su finalidad es indicar si se permitirá o no el uso de transacciones anidadas, siendo su valor por defecto `True`.

Como su propia denominación deja entrever, las transacciones anidadas son transacciones que se inician en el contexto de otra transacción ya en curso. De esta forma es posible dividir en varios pasos una operación de cierta complejidad, con una transacción global a toda la operación y transacciones individuales para las operaciones sobre la base de datos de cada paso.

El campo `EnableNested` de la propiedad `TxOptions`, de tipo booleano, determina si FireDAC permitirá o no el uso de transacciones anidadas. Su valor por defecto es `True`, habilitando esta funcionalidad. Las transacciones anidadas se controlan de forma manual, activándose en el momento en que se invoca al método `StartTransaction` en el contexto de una transacción en curso.

> **NOTA**
>
> *No todos los RDBMS contemplan el uso de transacciones anidadas. En aquellos casos en que dicha funcionalidad no esté disponible, FireDAC la simulará utilizando savepoints. Un savepoint es un recurso que permite volver al estado que tenía el conjunto de datos en un cierto instante.*

10.3.5 Conflictos de concurrencia

El uso de transacciones garantiza la integridad de la información alojada en la base de datos, según acaba de explicarse en los apartados previos. Una transacción, sin embargo, no resolverá los problemas de acceso concurrente a la información que puedan surgir. Estos se dan cuando dos o más solicitudes, procedentes normalmente de usuarios utilizando una aplicación que permite modificar datos, pretenden alterar una misma fila. Para facilitar el acceso concurrente a los datos, de forma que varios usuarios puedan trabajar simultáneamente con ellos, suele emplearse una estrategia de bloqueo optimista. Veamos en qué consiste y qué implica dicha estrategia.

La configuración de bloqueo utilizada por FireDAC viene determinada por la subpropiedad `LockMode` de la propiedad `UpdateOptions`. Su valor por defecto es `lmNone`, por lo que inicialmente no se bloquean las filas recuperadas por las consultas, permitiendo que múltiples usuarios operen sobre ellas. El principal inconveniente de esta configuración estriba en que no hay control alguno sobre el acceso concurrente a los datos, de forma que varias usuarios podrían enviar actualizaciones simultáneas a las tablas subyacentes y los cambios de uno sobrescribirían a los de otro, dando lugar a un escenario conocido habitualmente como *el último gana*. Debemos recordar que por defecto las actualizaciones sobre las tablas se producen utilizando en la cláusula `WHERE` sólo la clave primaria, por lo que el resto de los campos podrían haber sufrido alteraciones.

> **NOTA**
>
> *La configuración de bloqueo y otros aspectos del funcionamiento de FireDAC, utilizando propiedades como `UpdateOptions`, pueden establecerse en el componente `TFDConnection`, de forma que la configuración sea heredada por todos los conjuntos de datos enlazados a dicha conexión, o bien ajustarse individualmente para cada `TFDQuery`, `TFDTable`, etc.*

Los otros dos valores que puede tomar la citada propiedad `LockMode` son `lmOptimistic` y `lmPessimistic`, correspondiente a las estrategias de bloqueo optimista y pesimista, respectivamente. La primera intentará bloquear las filas afectadas justo antes de llevar a cabo su actualización, mientras que la segunda las bloqueará desde el mismo momento en que se solicitan los datos,

habitualmente utilizando la sintaxis `SELECT FOR UPDATE` de la sentencia `SELECT`. Optar por una estrategia u otra es una decisión que dependerá fundamentalmente de las necesidades de cada proyecto. El bloqueo de las filas de datos también se verá afectado por las siguientes dos propiedades:

- `LockPoint`: Esta propiedad determina cuándo se adquirirá el bloqueo de una fila de datos. Los valores posibles son `lpImmediate` y `lpDeferred`, siendo este último el usado por defecto. Con `lpImmediate` el bloqueo se establecerá en cuanto se active el modo de edición en el conjunto de datos. Usando `lpDeferred` el bloqueo se demorará hasta el momento en que vayan a enviarse los cambios al servidor.

- `LockWait`: Es una propiedad booleana que por defecto toma el valor `False`. Únicamente afecta a la estrategia de bloqueo pesimista. Con su valor por defecto se intentará bloquear la fila afectada y, en caso de que ya estuviese bloqueada, se lanzará una excepción. Dándole el valor `True` se esperaría a que el bloqueo en curso se levantase para adquirir el nuevo bloqueo.

Si el bloqueo no resulta posible, con independencia de cuándo quiera obtenerse, se generará una excepción. La probabilidad de que esto ocurra es mucho más alta cuando se utiliza bloqueo pesimista con bloqueo inmediato y sin espera. Es fácil comprobarlo con un sencillo ejercicio. Partiríamos, como siempre, de un nuevo proyecto multi-dispositivo con un formulario y un módulo de datos, reproduciendo a continuación los siguientes pasos:

1. Usando el Explorador de datos arrastramos una de las tablas de la base de datos `AdventureWorks` al formulario de datos, obteniendo así los componentes `TFDConnection` y `TFDQuery` adecuadamente configurados.

2. Seleccionamos el `TFDQuery` y cambiamos los valores de las propiedades `LockMode` y `LockPoint`, dando el valor `lmPessimistic` a la primera y `lpImmediate` a la segunda. Damos también el valor `True` a la propiedad `Active`.

3. Cambiamos al formulario e introducimos en él un control `TGrid`, vinculándolo con el `TFDQuery`. De esta forma podremos editar el contenido de la tabla.

EJEMPLO 10.1 `CSTransaction.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSTransaction`. Deberás ajustar el componente `TFDConnection` para conectar con el servidor SQL Server y abrir la base de datos `AdventureWorks`.

Tras compilar el programa tendremos que ejecutar al menos dos instancias del mismo accediendo a la misma base de datos. El hecho de que cada instancia se ejecute en el mismo equipo o en máquinas distintas es indiferente. Tras comenzar a editar una de las filas en una instancia, al intentar hacer lo mismo en otra obtendremos un error como el mostrado en la Figura 10.2.

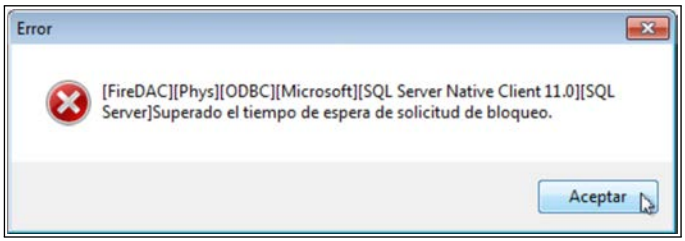


Figura 10.2 NO ES POSIBLE OBTENER UN BLOQUEO SOBRE LA FILA

Ante esta situación podemos proceder básicamente de dos maneras:

- Controlar la excepción e indicar al usuario que en ese momento la fila sobre la que quiere trabajar no está accesible, recomendándole intentarlo pasado un cierto tiempo.

- Asignar el valor `True` a la propiedad `LockWait` del `TFDQuery`. Esto provocará que el programa espere hasta poder obtener el bloqueo, mostrando mientras tanto el cursor indicativo de espera.

NOTA

En este ejercicio, al solicitarse el cambio al modo de edición del conjunto de datos desde la propia interfaz de usuario (el control `TGrid`), el proceso de espera tendrá como resultado el bloqueo temporal del programa. En cuanto la instancia que estaba editando la fila libere el bloqueo la instancia que estaba esperando volverá a responder.

10.4 Notificación de cambios

Al trabajar en un entorno cliente/servidor es muy probable que los datos que está mostrando la interfaz a un usuario estén siendo modificados por otra persona. Ya hemos abordado cómo controlar el acceso concurrente, así como el nivel de aislamiento ofrecido por las transacciones al acceder a datos de forma simultánea desde varios clientes.

Además de evitar sobrescribir los cambios aplicados por otros usuarios, bloqueando si es necesario las filas sobre las que se va a operar, habitualmente también resulta útil obtener dichos cambios a fin de actualizar la información que está mostrándose al usuario. Con este fin puede utilizarse un mecanismo de notificación de cambios, consistente en los siguientes pasos:

1. El cliente comunica al servidor los cambios sobre los que está interesado en ser notificado.
2. El RDBMS registra la solicitud activando los elementos necesarios en el servidor, generalmente en forma de desencadenadores (*triggers*).
3. Cuando el el desencadenador se dispara el servidor notifica al cliente que se ha producido un cambio.

En esta sección se describe cómo utilizar este mecanismo de notificación con un ejemplo concreto, utilizando la base de datos SQL Server `AdventureWorks`. El pilar fundamental del sistema de notificación de cambios es el componente `TFDEventAlerter`.

10.4.1 Configuración de la solicitud

El componente `TFDEventAlerter` se asocia a una conexión a base de datos mediante la propiedad `Connection`. Esta hará referencia al `TFDConnection` usado para realizar las consultas y actualizaciones de datos, tal y como se aprecia en la Figura 10.3. La configuración de buena parte de las propiedades de este componente, incluyendo `Options` y `Names`, son dependientes de cada RDBMS.

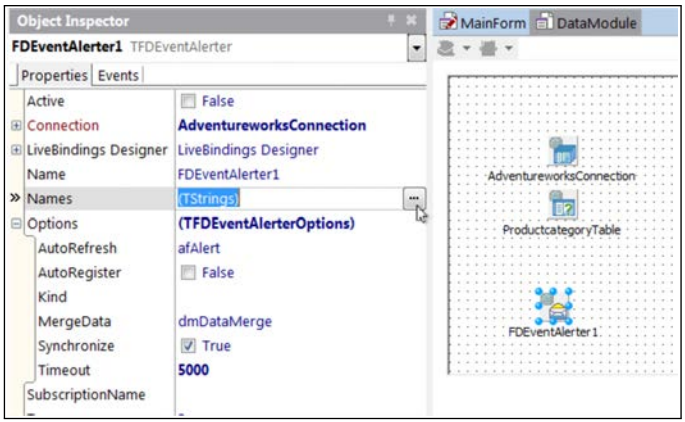


Figura 10.3 PROPIEDADES DEL COMPONENTE `TFDEventAlerter`

Con la propiedad `Options.Kind` se establece el tipo de mecanismo de notificación a emplear. Si la dejamos en blanco se utilizará el mecanismo[1] por defecto del RDBMS. La propiedad `Options.Synchronize` la dejaremos habitualmente a `True`, ya que el sistema de notificaciones se ejecuta en el cliente en un hilo (*thread*) independiente, siendo precisa la sincronización con el hilo principal para poder acceder de forma segura a la interfaz de usuario.

Mediante la propiedad `Names` se facilitarán los parámetros necesarios para indicar los eventos en que estamos interesados. El formato de esta propiedad

[1] En la página `http://docwiki.embarcadero.com/RADStudio/XE8/en/Database_Alerts_(FireDAC)` podemos encontrar la lista de valores para la propiedad `Kind` según el RDBMS con el que se trabaje.

también depende de cada RDBMS. En algunos casos es suficiente con indicar el nombre de la tabla cuyos cambios se quieren vigilar, mientras que en otros es necesario asignar un nombre a cada evento y especificar la consulta cuyos resultados se supervisarán.

El componente puede facilitar una notificación al recibir una señal procedente del RDBMS y también cuando haya transcurrido un cierto tiempo sin recibir dicho evento. El lapso de tiempo a esperar es el establecido por la propiedad `Timeout`, cuya unidad de medida es el milisegundo.

10.4.2 Registro con el servidor y eventos

Una vez establecida la configuración es necesario registrar la solicitud con el servidor. Para ello tenemos fundamentalmente dos opciones: utilizar la propiedad `Active`, de tipo booleano, o buen invocar en ejecución a los métodos `Register` y `Unregister`. Dar el valor `True` a la propiedad `Active` equivale a llamar al método `Register`. Este no tendrá ningún efecto en caso de que dicha propiedad ya esté activada.

Tras registrar la solicitud en el servidor, cada vez que este detecte cambios en los objetos indicados por la propiedad `Names` se procederá a realizar una notificación. Esta también puede producirse si transcurre el tiempo indicado por `Timeout`. En el primer caso se lanzará el evento `OnAlert` y en el segundo el evento `OnTimeout`.

> **NOTA**
>
> *No todos los RDBMS cuentan con un mecanismo de notificación de cambios automático. En el caso de InterBase y SQL Server sí existe, pero para otros servidores de datos, como Oracle o Firebird, es preciso crear manualmente en el servidor los desencadenadores apropiados a fin de que el RDBMS envíe una señal al cliente.*

Lo habitual es que al recibir la notificación de cambio el programa solicite la actualización del contenido del `TFDQuery` o `TFDTable` afectado, usando para ello su método `Refresh`.

10.4.3 Actualización automática de datos

El uso más habitual para las notificaciones de cambios es la actualización de los datos mostrados al usuario, refrescando el contenido del conjunto de datos pertinente. Por ello en la versión XE8 de Delphi se han introducido como novedad en el componente `TFDQuery` las propiedades `ChangeAlerter` y `ChangeAlertName` (véase la Figura 10.4). La primera enlaza el `TFDQuery` con el `TFDEventAlerter`. Con la segunda es posible indicar el nombre de un evento que, por defecto, será el nombre de la tabla de la que toma la consulta los datos.

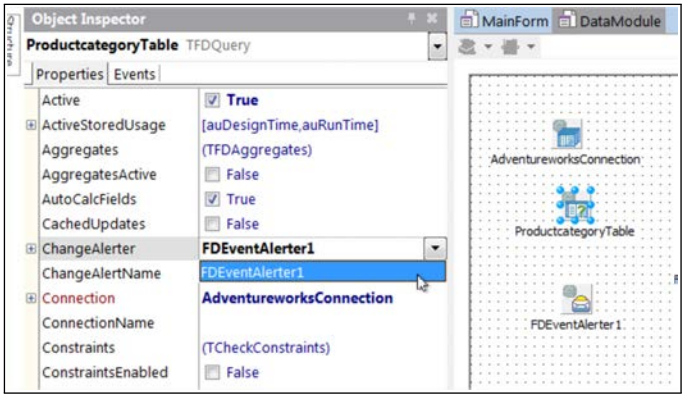


Figura 10.4 NUEVAS PROPIEDADES EN EL COMPONENTE `TFDQUERY`

Este mecanismo de actualización automática del contenido del `TFDQuery` cuando cambian los datos en el RDBMS funciona con InterBase y SQL Server. Para el resto de servidores de datos será necesario configurar e l componente `TFDEventAlerter` según lo indicado en el apartado previo, así como crear los desencadenadores adecuados en la base de datos.

10.4.4 Transacciones y notificaciones en la práctica

Para finalizar vamos a servirnos de un ejercicio que nos permita experimentar tanto con el control manual de transacciones como con las notificaciones de cambios. El objetivo es editar simultáneamente en dos o más instancias de un programa los mismos datos, de forma que los cambios en una copia se reflejen en las demás y, opcionalmente, podamos controlar manualmente las transacciones.

Como es habitual, iniciamos un nuevo proyecto de tipo multi-dispositivo y le agregamos un módulo de datos. A continuación reproducimos los siguientes pasos:

1. Tomamos del Explorador de datos la tabla `ProductCategory` de la base de datos `AdventureWorks` y la arrastramos hasta el módulo de datos, configurando así tanto la conexión como el `TFDQuery` que representa al conjunto de datos.

2. Agregamos un componente `TFDEventAlerter`, abrimos el editor asociado a su propiedad `Names` e introducimos las líneas mostradas en la Figura 10.5. Las entradas `SERVICE` y `QUEUE` con el valor `?` crearán en el RDBMS un servicio y una cola con un nombre único para supervisar los cambios en la consulta indicada por la tercera línea. También modificaremos la propiedad `Timeout` asignándole el valor `5000`.

3. Nos serviremos de los eventos `OnCreate` y `OnDestroy` del formulario para activar y desactivar el registro de la solicitud de notificación en el servidor, sirviéndonos para ello de los métodos `Register` y `Unregister` tal y como se muestra en el siguiente listado:

```
1 procedure TdmChangeNotification.DataModuleCreate(Sender: ↙
      ↳ TObject);
2 begin
3   FDEventAlerter1.Register;
4 end;
5
6 procedure TdmChangeNotification.DataModuleDestroy(Sender:↙
      ↳  TObject);
```

```
7 begin
8   FDEventAlerter1.Unregister;
9 end;
```

Listado 10.1 Activación y desactivación del `TFDEventAlerter`

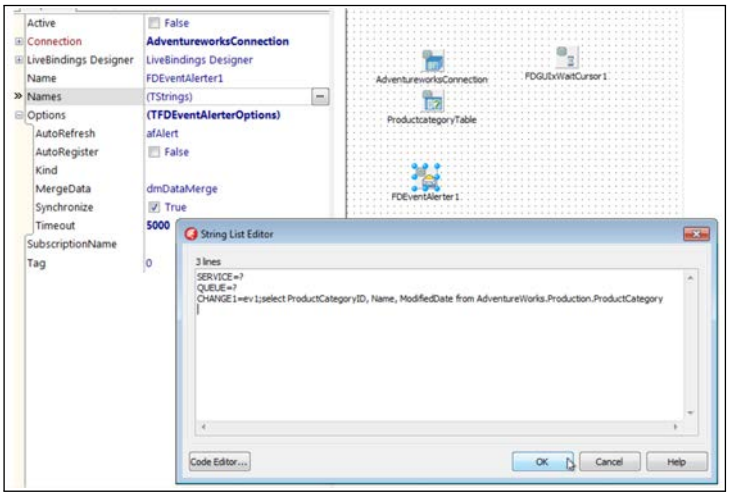


Figura 10.5 CONFIGURACIÓN DEL EVENTO A VIGILAR EN LA BASE DE DATOS

4. El siguiente paso será responder a los eventos `OnAlert` y `OnTimeout` del componente `TFDEventAlerter`. En ambos casos de la misma forma: actualizando el contenido del `TFDQuery` y forzando la revisualización en el `TGrid`. El listado siguiente muestra el código del primero de los eventos, el segundo sería idéntico:

```
1 procedure TdmChangeNotification.FDEventAlerter1Alert(
2   ASender: TFDCustomEventAlerter; const AEventName: ↙
      ↳ string;
3   const AArgument: Variant);
4 begin
5   ProductcategoryTable.Refresh;
```

```
6    frmMain.Grid1.RealignContent;
7  end;
```

Listado 10.2 Actualizamos los datos al recibir el evento

5. Ahora cambiamos al formulario, agregamos una referencia al módulo de datos e introducimos en el mismo un `TPanel` con tres `TButton`, en la parte superior, y un `TGrid` ocupando el resto del espacio disponible, tal y como se muestra en la Figura 10.6. Los botones de confirmación y cancelación de transacción estarán inicialmente inactivos. Enlazamos el `TGrid` con el `TFDQuery` existente en el módulo de datos.

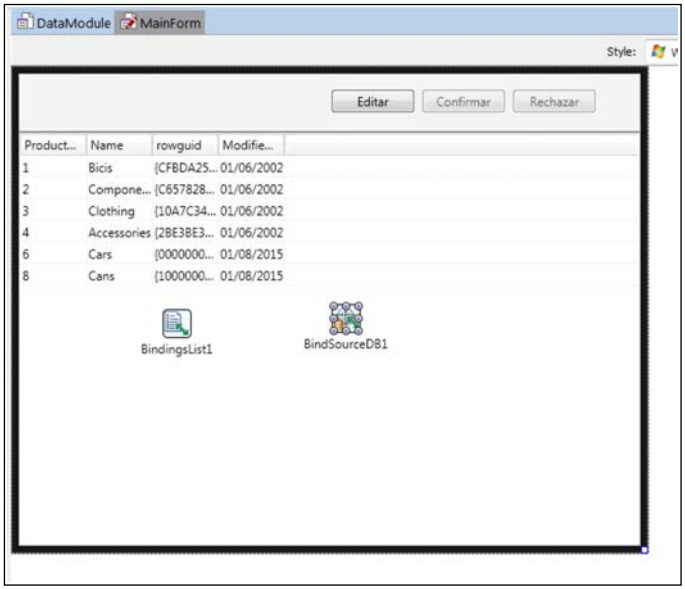


Figura 10.6 FORMULARIO PARA CONTROLAR TRANSACCIONES Y EDITAR LAS CATEGORÍAS

6. Finalmente tendremos que escribir el código asociado a cada uno de los botones incluidos en el formulario. Los datos podrán editarse directamente en la cuadrícula usando transacciones automáticas, pero si se hace clic en el

primer botón se pondrá en marcha una transacción manual. Esta podrá confirmarse o cancelarse con los otros dos botones. De esta forma podremos experimentar comprobando cómo los cambios no se notifican a las demás copias del programa en ejecución hasta que la transacción es confirmada. El código asociado a los botones es el siguiente:

```
1  procedure TfrmMain.btnEditarClick(Sender: TObject);
2  begin
3    dmChangeNotification.AdventureworksConnection.↙
         ↳ StartTransaction;
4    btnEditar.Enabled := False;
5    btnConfirmar.Enabled := True;
6    btnRechazar.Enabled := True;
7  end;
8
9  procedure TfrmMain.btnConfirmarClick(Sender: TObject);
10 begin
11   dmChangeNotification.AdventureworksConnection.Commit;
12   btnEditar.Enabled := True;
13   btnConfirmar.Enabled := False;
14   btnRechazar.Enabled := False;
15 end;
16
17 procedure TfrmMain.btnRechazarClick(Sender: TObject);
18 begin
19   dmChangeNotification.AdventureworksConnection.Rollback;
20   btnEditar.Enabled := True;
21   btnConfirmar.Enabled := False;
22   btnRechazar.Enabled := False;
23 end;
```

Listado 10.3 Métodos asociados a cada botón de la interfaz

Para comprobar el sistema de notificaciones deberemos ejecutar varias instancias de esta aplicación. Acto seguido podemos introducir cambios en una de ellas, modificando una de las filas de la cuadrícula, y comprobar cómo al cambiar de fila dichos cambios se reflejan en los demás. En esta situación estarán

en funcionamiento las transacciones automáticas. La otra alternativa es iniciar una transacción, con el único botón activo, y después realizar cambios en una o más filas. Estos no se confirmarán hasta que usemos el botón correspondiente, por ello no aparecerán en las demás instancias del programa de manera inmediata. Además podríamos usar el tercer botón para cancelar los cambios, forzando la recuperación del estado anterior.

EJEMPLO 10.2 `CSChangeNotification.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSChangeNotification`. Deberás ajustar el componente `TFDConnection` para conectar con el servidor SQL Server y abrir la base de datos `AdventureWorks`.

10.5 Resumen

En este capítulo se ha explicado la diferencia entre el sistema de control de transacciones automático, usado por defecto en FireDAC, y la gestión manual de las mismas que aporta mayor control y flexibilidad. También se han descrito las distintas estrategias de bloqueo de filas a la hora de actuar sobre ellas, así como la forma en que podemos obtener notificaciones cuando la información cambia en el servidor de datos.

El siguiente capítulo aborda otro tema de interés cuando se trabaja en un entorno cliente/servidor: la visualización y edición de los datos sin contar con una conexión permanente al servidor de datos.

Capítulo 11

TRABAJAR SIN CONEXIÓN AL RDBMS

En contraposición a las aplicaciones que operan de forma continua con el RDBMS, enviando datos o solicitándolos con cierta frecuencia, hay programas que sólo necesitan la conexión con el servidor de datos de forma esporádica. En estos casos es posible trabajar temporalmente sin conexión, reduciendo así la carga del servidor y mejorando su escalabilidad, al poder atender a un mayor número de usuarios con los mismos recursos.

Como ya sabemos (véase el Capítulo 7) los componentes que mantienen los conjuntos de datos en FireDAC, como es el caso de `TFDQuery`, están internamente diseñados para operar con los datos en memoria, sin necesidad de contar

con una conexión permanente a la base de datos. Esto permite desarrollar aplicaciones que se conectan puntualmente al RDBMS, recuperan la información que necesitan y cierran dicha conexión, trabajando sobre los datos de forma local. La comunicación con el servidor de datos se restablece puntualmente, cuando es preciso transferir cambios o demandar información adicional.

Este capítulo describe la configuración y el procedimiento a seguir en aplicaciones que no requieren una conexión permanente con el RDBMS para realizar su trabajo. Es un objetivo para el que se precisan dos funcionalidades: el cierre de la conexión sin perder los datos alojados en los componentes y el almacenamiento temporal de los cambios aplicados a esos datos.

11.1 Modo desconectado de FireDAC

Cuando se da el valor `True` a la propiedad `Active` de un conjunto de datos, por un ejemplo un `TFDQuery`, o alternativamente se invoca a su método `Open`, se usa el vínculo con la conexión para abrir esta si es necesario y, a continuación, recuperar los datos establecidos por la consulta de selección. La forma en que se recuperan los datos vendrá determinada por algunas de las propiedades descritas en la Sección 9.2.3. Desde ese momento tanto la conexión como los conjuntos de datos enlazados a ella se mantienen abiertos, haciendo así posible la transferencia al servidor de los cambios que se apliquen y, si fuera preciso, la recuperación de datos adicionales.

Habitualmente cada conjunto de datos en el cliente mantendrá abierto un cursor en el servidor, asociado al comando de selección de datos, facilitando así la navegación por las filas de la tabla. Si se cerrase la conexión, por tanto, también se cerrarían los cursores de base de datos, así como los comandos y conjuntos de datos. El resultado sería que todos los controles de interfaz enlazados a datos quedarían vacíos, no siendo posible el acceso a la información. En principio esto nos obligaría a mantener una conexión constante con el servidor de datos.

Para seguir operando sobre los datos tras haber cerrado la conexión con el RDBMS tendremos que activar el modo desconectado, una característica de FireDAC que reducirá la carga del servidor de datos y permitirá a los clientes disponer de los datos sin mantener activa una conexión que, dependiendo de la localización y recursos, puede resultar cara o difícil.

11.1.1 Conectar/Desconectar de la base de datos

El modo desconectado de FireDAC consiste en mantener abiertos los conjuntos de datos, alojando la información necesaria en memoria, incluso después de haber cerrado los comandos, cursores y la conexión con la base de datos. En lugar de dar el valor `False` a la propiedad `Connected` del `TFDConnection` o llamar al método `Close`, usaremos la propiedad `Offlined` o bien el método `Offline`, miembros ambos de `TFDConnection`. Asignar el valor `True` a dicha propiedad equivale a llamar al método `Offline`.

En el momento en que se activa el modo desconectado, y justo antes de proceder al cierre efectivo de la conexión con la base de datos, cada uno de los conjuntos de datos enlazados al `TFDConnection` será notificado acerca de la operación que va a realizarse. En respuesta, cada conjunto de datos reaccionará dependiendo del valor que se hubiese asignado al campo `AutoFetchAll` de la propiedad `FetchOptions`. Hay tres posibilidades que corresponden a los tres valores válidos para dicho campo:

- `afAll`: Este valor es el usado por defecto. Provoca que el conjunto de datos recupere todas las filas generadas por el comando asociado antes de pasar a operar en modo desconectado.

- `afTruncate`: Utilizando esta configuración el conjunto de datos pasará al modo desconectado de inmediato, a pesar de que no disponga de todas las filas.

- `afDisable`: Si se asigna este valor al citado campo el conjunto de datos no permitirá trabajar en modo desconectado, por lo que generará una excepción si este se activa.

Asumiendo que finalmente se active el modo desconectado, automáticamente se cerrará la conexión con la base de datos y la propiedad `Connected`, que indica el estado de la conexión, pasará a ser `False`. Para reactivar la conexión bastará con dar el valor `False` a la propiedad `Offlined` o, si lo preferimos, llamar al método `Online` del `TFDConnection`. En ese momento se reabrirá la conexión pero sin necesidad de ejecutar los comandos de recuperación de información, ya que los conjuntos de datos tienen el contenido apropiado.

11.1.2 Reconexión automática

Una vez que se ha activado el modo desconectado, cabe preguntarse qué ocurrirá en caso de que se precise ejecutar una operación que demanda una conexión con el RDBMS. ¿Qué pasa si, por ejemplo, modificamos una fila de datos en una cuadrícula y el `TFDQuery` conectado a la misma llama al método `Post` para enviar los cambios? La respuesta es que dependerá del valor que contenga una propiedad booleana: `AutoConnect`. Esta es una subpropiedad de la propiedad `ResourceOptions` y puede ser configurada de forma individual, para cada `TFDQuery`, `TFDTable`, etc., o bien de manera global en el `TFDConnection`.

> **EJEMPLO 11.1** `CSOffline.dproj`
>
> Puedes encontrar este proyecto completo en la carpeta `CSOffline`. Deberás ajustar el componente `TFDConnection` para conectar con el servidor SQL Server y abrir la base de datos `AdventureWorks`.

Si la citada propiedad tiene el valor `True`, ese es su estado por defecto, la conexión con el RDBMS se restablecerá automáticamente cuando sea necesario, por ejemplo en la situación que acaba de exponerse. Esto implica que `Connected` volverá a tomar el valor `True` mientras que `Offlined` retornará a su valor `False` inicial. Tras completar la operación demandada no se volverá al modo desconectado, sino que la conexión permanecerá activa.

Dando el valor `False` a la propiedad `AutoConnect` denegaremos la conexión automática cuando sea preciso. Si se da una situación como la citada, el componente generará una excepción al no poder completar su trabajo.

Podemos comprobar las distintas casuísticas de trabajo con el modo desconectado con un sencillo ejercicio. El nuevo proyecto, de tipo multi-dispositivo, contará con un módulo de datos al que llevaremos la tabla de categorías de producto de la base de datos `AdventureWorks`, como en ejercicios de capítulos previos. Modificaremos la propiedad `Filter` del `TFDQuery` asignándole el valor `name LIKE 'C%'`. El objetivo es filtrar las filas mostradas simplemente mo-

dificando la propiedad booleana `Filtered`, sin necesidad de cambiar y volver a ejecutar la consulta.

En el formulario del proyecto insertaremos los controles que se pueden apreciar en la Figura 11.1, atendiendo a la siguiente configuración:

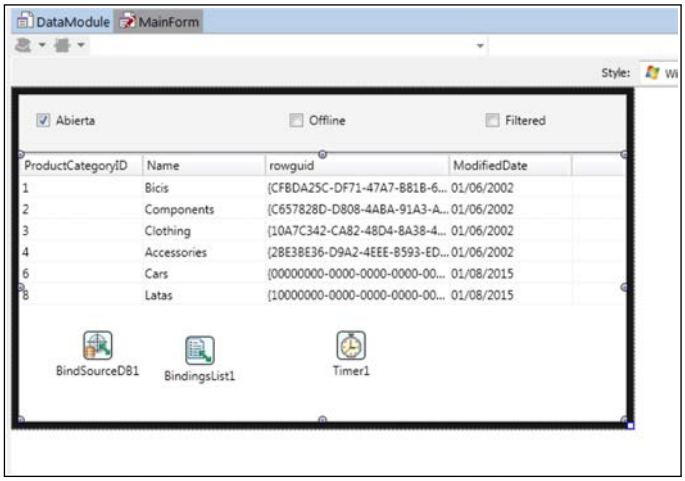


Figura 11.1 FORMULARIO PARA PROBAR EL MODO DESCONECTADO

- `TPanel`: Ajustado a la parte superior del formulario. Servirá como contenedor de tres `TCheckBox`.
- `TGrid`: Ocupará el resto del espacio disponible en el formulario. Lo enlazaremos con el `TFDQuery` existente en el módulo de datos.
- `TCheckBox`: Servirán para controlar el estado de la conexión. El primero de ellos tendrá el valor `True` en su propiedad `IsChecked`, denotando así que la conexión está inicialmente abierta.
- `TTimer`: Servirá para actualizar periódicamente la propiedad `IsChecked` de los tres `TCheckBox` a fin de que reflejen el estado de la conexión a la base de datos.

El primer TChecBox (más a la izquierda) controlará el estado de la conexión a la base de datos. Al hacer clic sobre él se ejecutará el código mostrado a continuación. Esté activa o desactiva la conexión y en caso necesario, si lo que se ha hecho es activar la conexión, abre el TFDQuery recuperando su contenido y mostrándolo en la cuadrícula:

```
1 procedure TfrmMain.CheckBox1Change(Sender: TObject);
2 begin
3   with dmEmployee.AdventureworksConnection do
4   begin
5     Connected := CheckBox1.IsChecked;
6     if Connected then
7       dmEmployee.ProductcategoryTable.Open;
8   end;
9 end;
```

Listado 11.1 Control de la conexión a la base de datos

Al hacer clic en este control no solo se cerrará la conexión con la base de datos, sino también los comandos y conjuntos de datos asociados. Esto implica que el TFDQuery perderá su contenido y, por tanto, la cuadrícula quedará vacía.

El evento OnChange del segundo TCheckBox tendrá asociado el código siguiente:

```
1 procedure TfrmMain.CheckBox2Change(Sender: TObject);
2 begin
3   dmEmployee.AdventureworksConnection.Offlined := CheckBox2.↙
      ↳ IsChecked;
4 end;
```

Listado 11.2 Control para activar/desactivar el modo desconectado

Lo que se hace es modificar el valor de la propiedad Offlined, activando o desactivando el modo desconectado. Al activarlo podremos observar cómo el primer TCheckBox se desmarca, indicando que se ha cerrado la conexión con la base de datos. A pesar de ello el TFDQuery no se cerrará, por lo que la cuadrícula seguirá mostrando su contenido.

El tercer `TCheckBox` se limitará a cambiar el valor de la propiedad `Filtered` del `TFDQuery`, filtrando así las filas mostradas en la cuadrícula. El código asociado al evento `OnChange` es el mostrado a continuación:

```
1 procedure TfrmMain.CheckBox3Change(Sender: TObject);
2 begin
3   dmEmployee.ProductcategoryTable.Filtered := CheckBox3.↙
      ↳ IsChecked;
4 end;
```

Listado 11.3 Control para filtrar las filas mostradas

El objetivo de este control es mostrar cómo a pesar de estar en modo desconectado es posible seguir trabajando con el `TFDQuery`, por ejemplo filtrando las filas.

Finalmente tenemos el código asociado al evento `OnTimer` del componente `TTimer`, que será el mostrado a continuación:

```
1 procedure TfrmMain.Timer1Timer(Sender: TObject);
2 begin
3   with dmEmployee do
4   begin
5    CheckBox1.IsChecked := AdventureworksConnection.Connected↙
       ↳ ;
6    CheckBox2.IsChecked := AdventureworksConnection.Offlined;
7    CheckBox3.IsChecked := ProductcategoryTable.Filtered;
8   end;
9 end;
```

Listado 11.4 Actualización periódica de los `TCheckBox`

Si tras haber activado el modo desconectado introducimos algún cambio en los datos mostrados en la cuadrícula, forzando así una comunicación el servidor, el código anterior se encargará de reflejar el nuevo estado de la conexión, que se reactivará automáticamente. Si lo deseamos podemos desactivar la conexión automática para verificar cómo se produce una excepción ante la misma acción.

11.2 Actualizaciones por lotes

El esquema de trabajo por defecto de FireDAC, como ya sabemos, incluye la recuperación de datos actualizables a demanda a partir de las consultas y la transferencia de los cambios de manera inmediata al RDBMS. Con la configuración adecuada del modo desconectado, que fuerza la recuperación de todas las filas en cada conjunto de datos antes de desconectar de la base de datos, la recuperación de datos a demanda ya no es necesaria. Queda, no obstante, la necesidad de enviar las actualizaciones de datos a la base de datos: inserción, modificación y borrado de filas de datos.

Mediante una técnica denominada *cached updates* FireDAC puede almacenar localmente todos los cambios efectuados a los datos, llevando a cabo actualizaciones por lotes en los momentos en que se cuente con una conexión con el servidor. Existen dos modos de actualizaciones por lotes: centralizado y descentralizado. En esta sección se explica en qué consisten las actualizaciones por lotes y se enumeran las propiedades y componentes que es necesario configurar según el modo que elijamos utilizar.

11.2.1 Activación de la actualización por lotes

Todos los derivados de `TFDDataSet`, entre los que se cuentan los componentes `TFDQuery` y `TFDTable`, heredan una propiedad booleana llamada `CachedUpdates`. Esta toma por defecto el valor `False`. Al darle el valor `True` se modifica el comportamiento del componente, de forma que las llamadas a métodos como `Edit`, `Post`, `Insert` y `Delete` no demandan una comunicación inmediata con el RDBMS, sino que generan entradas en un registro de cambios[1] mantenido por el propio componente.

Tras activar el registro de cambios en un conjunto de datos, y efectuar distintas operaciones sobre él, tenemos dos alternativas:

- Invocar al método `ApplyUpdates` en el momento en que volvamos a disponer de una conexión con la base de datos, enviándole los comandos necesarios para aplicar todos los cambios.

[1] Este registro queda almacenado en la propiedad `Delta` del componente.

- Descartar todas las operaciones de cambio mediante el método `CancelUpdates`, dejando el conjunto de datos tal y como estaba al abrirlo.

Estas operaciones las llevaríamos a cabo antes de cerrar el conjunto de datos explícitamente, dando el valor `False` a su propiedad `Active` o llamando al método `Close`, o bien implícitamente, al cerrar una conexión abierta con la base de datos.

> **NOTA**
>
> *El cierre del conjunto de datos y la asignación del valor `False` a la propiedad `CachedUpdates` del mismo implican la ejecución del método `CancelUpdates`, por lo que todos los cambios efectuados se perderían.*

La actualización por lotes resulta útil no solo en el contexto esbozado en la primera parte de este capítulo, con aplicaciones que operan de forma desconectada respecto al servidor de datos, sino también en aplicaciones estándar que, a pesar de contar con una conexión permanente al RDBMS, usan esta técnica para reducir el número de transferencias de datos y de transacciones, mejorando el rendimiento global. Como contrapartida, ha de tenerse en cuenta que la posibilidad de que surjan conflictos al demorar la actualización de los datos se incrementa, ya que múltiples usuarios pueden haber obtenido las mismas filas y querer introducir cambios en ellas en el lapso de tiempo en que se opera sin conexión. Además, los cambios hechos en uno de los clientes no serán visibles para el resto hasta que no se envíen las actualizaciones al servidor.

11.2.2 Deshacer cambios hechos a los datos

Una interesante ventaja adicional al activar la actualización por lotes es la posibilidad de deshacer cambios que aún no han sido transmitidos a la base de datos. Esta acción consiste en eliminar del registro local de cambios las operaciones que afectan a un cierto registro. Para ello usaremos uno de los dos métodos siguientes:

- `RevertRecord`: Elimina los cambios que afectan a la fila actual del conjunto de datos, dejándola tal y como estaba cuando se recuperó de la base de datos.
- `UndoLastChange`: Activa la última fila que haya sufrido cambios y los deshace, dejando los datos en su estado anterior.

Otra vía para deshacer cambios, no a una fila sino de forma global en el conjunto de datos, consiste en leer y modificar el contenido de la propiedad `SavePoint` del componente. Al leerla se obtiene un número que identifica el estado del conjunto de datos en ese instante. Dicho valor puede utilizarse posteriormente para devolver el conjunto de datos a ese estado, por ejemplo si tras realizar un conjunto de operaciones el usuario quiere deshacerlas o se produce algún tipo de fallo.

NOTA

Para determinar si un conjunto de datos tiene cambios pendientes de enviar al servidor podemos consultar la propiedad `UpdatesPending`. Si esta tiene el valor `True`, la propiedad `ChangeCount` indica el número de cambios que hay en el registro local.

11.2.3 Envío de los cambios pendientes

Al trabajar con un conjunto de datos a cuya propiedad `CachedUpdates` se ha asignado el valor `True` hemos de tener en cuenta que el envío de los cambios pendientes es responsabilidad nuestra, no algo que vaya a ocurrir automáticamente. Esta operación consta habitualmente de tres pasos:

1. Llamada al método `ApplyUpdates`. Este se encargará de recorrer el registro de cambios local, almacenado en la propiedad `Delta`, e ir enviando al servidor los comandos de actualización pertinentes para que tengan su reflejo en la base de datos.
2. Llamada al método `Reconcile`. Tras enviar los cambios al servidor este método se encarga de verificar el resultado generado por cada comando,

comprobando si la respectiva actualización ha podido aplicarse o no. Por cada actualización fallida generará un evento `OnReconcileError` (véase el siguiente apartado de esta sección).

3. Llamada al método `CommitUpdates`. Este método se encarga de eliminar el registro local asociado al conjunto de datos, de forma que el estado actual de las filas pasa a ser definitivo.

El método `ApplyUpdates` devolverá un valor numérico indicando el número de errores que se han producido durante el proceso de aplicación de cambios en la base de datos. La aplicación debe comprobar este valor, ya que no siempre se genera una excepción en caso de que algunas de las entradas en el registro de cambios no puedan ser efectivamente aplicadas en la base de datos. Las llamadas a los métodos `Reconcile` y `CommitUpdates` se producen automáticamente para ciertos componentes, como las tablas en memoria, pero no para `TFDQuery` o `TFDTable` cuando se opera en modo no centralizado.

En cualquier momento podemos filtrar el contenido del conjunto de datos a fin de determinar qué filas permanecen con sus valores originales, cuáles han sufrido cambios, cuáles son nuevas o se han eliminado, etc. Para ello usaremos la propiedad `FilterChanges` del conjunto de datos, asignándole un conjunto en el que pueden aparecer las siguientes constantes:

- `rtUnmodified`: Se incluyen las filas que no han sufrido cambios.
- `rtModified`: Se incluyen las filas cuyo contenido ha sido modificado.
- `rtInserted`: Se incluyen las nuevas filas añadidas al conjunto de datos.
- `rtDeleted`: Se incluyen las filas que han sido eliminadas.
- `rtHasErrors`: Se incluyen las filas que tienen errores de actualización.

El valor asignado por defecto a la propiedad `FilterChanges` es el conjunto `[rtModified,rtInserted,rtUnmodified]`, de forma que son mostradas las filas sin cambios, las actualizadas y las nuevas, pero no las eliminadas.

Veamos con un sencillo pero demostrativo ejercicio cómo utilizar los métodos `ApplyUpdates` y `CancelUpdates`, así como el filtrado de filas según su estado a fin de revisar los cambios y algunas propiedades más de las descritas hasta ahora.

EJEMPLO 11.2 `CSCachedUpdates.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSCachedUpdates`. Deberás ajustar el componente `TFDConnection` para conectar con el servidor SQL Server y abrir la base de datos `AdventureWorks` o bien modificar la conexión para usar otra base de datos distintas.

Partiendo del proyecto vacío, agregaremos un módulo de datos en el que insertaremos los componentes necesarios para recuperar el contenido de la tabla de categorías de producto, como en otros casos. Modificaremos las siguientes propiedades del componente `TFDQuery`:

- `CachedUpdates`: Le asignaremos el valor `True` a fin de activar las actualizaciones por lotes.
- `UpdateOptions.UpdateMode`: Elegiremos de la lista desplegable el valor `upWhereChanged`, de forma que al actualizar se utilicen la clave primaria y los campos que sufran cambios.
- `Active`: También lo pondremos a `True` a fin de activar la conexión y recuperar los datos.

A continuación compondremos la interfaz de usuario de la aplicación. Esta se dividirá en tres secciones claramente diferenciadas representadas por los siguientes componentes:

- `TPanel`: Ajustado a la parte superior del formulario. Servirá para alojar un `TCheckBox` y dos `TButton`.
- `TBindNavigator`: Ajustado a la parte inferior del formulario. Facilitará las operaciones de inserción y eliminación de filas.
- `TGrid`: Ocupará el resto del espacio disponible en el formulario, permitiendo mostrar y editar el contenido del `TFDQuery` con el que lo enlazaremos.

En la Figura 11.2 se muestra el formulario en el diseñador con todos los elementos que hemos de introducir en él, así como el enlace entre los datos y los controles en la parte inferior.

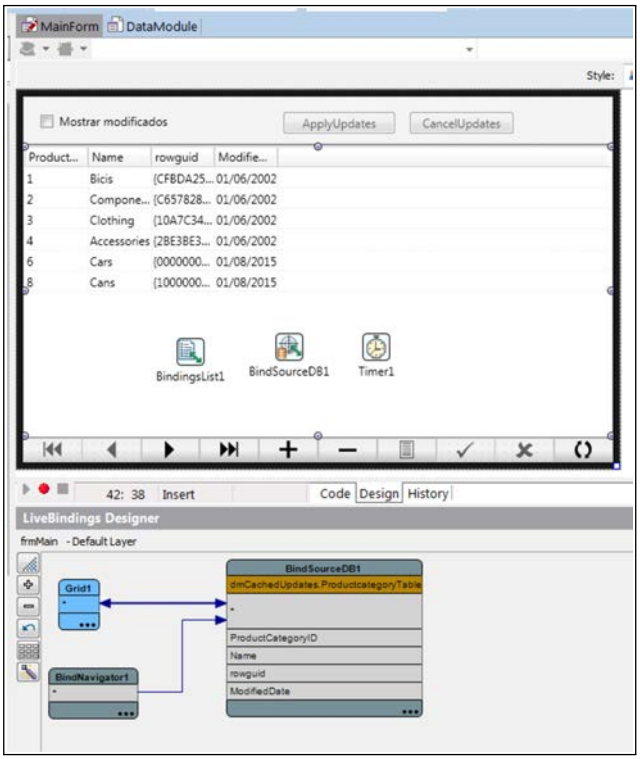


Figura 11.2 ELEMENTOS EN LA INTERFAZ DE USUARIO

Los demás controles a introducir en la interfaz de usuario, y su configuración, son los indicados a continuación:

- `TCheckBox`: El objetivo de este control será activar/desactivar el filtrado de filas en la cuadrícula, de forma que se alterne entre la vista normal y una alternativa que muestra únicamente filas modificadas, agregadas o eliminadas. El evento `OnChange` de este control tendrá asociado el código si-

guiente, encargado de modificar la propiedad `FilterChanges` del componente `TFDQuery` según la información que quiere mostrarse:

```
1  procedure TfrmMain.cbShowChangesChange(Sender: TObject);
2  begin
3    with dmCachedUpdates.ProductcategoryTable do
4      if cbShowChanges.IsChecked then
5        FilterChanges := [rtModified,rtInserted,rtDeleted]
6      else
7        FilterChanges := [rtModified,rtInserted,↙
             ↳ rtUnmodified];
8  end;
```

Listado 11.5 Cambiamos el filtrado de cambios del `TFDQuery`

- `TButton`: Estos dos botones serán los encargados de invocar a los métodos `ApplyUpdates` y `CancelUpdates` a demanda, lo cual nos permitirá realizar distintas pruebas. El código asociado a los respectivos eventos `OnClick` es el siguiente:

```
1  procedure TfrmMain.btnApplyUpdatesClick(Sender: TObject);
2  var
3    nErrors: Integer;
4  begin
5    nErrors := dmCachedUpdates.ProductcategoryTable.↙
         ↳ ApplyUpdates;
6    if nErrors <> 0 then
7      ShowMessage(IntToStr(nErrors) + ' errores');
8  end;
9
10 procedure TfrmMain.btnCancelUpdatesClick(Sender:TObject);
11 begin
12   dmCachedUpdates.ProductcategoryTable.CancelUpdates;
13 end;
```

Listado 11.6 Envío y cancelación de cambios pendientes

- `TTimer`: La finalidad de este componente es comprobar periódicamente si hay o no cambios pendientes de enviar al servidor, actualizando en concordancia el estado de los dos `TButton` y el título mostrado en el primero de ellos a fin de incluir el número de cambios p endientes. El código asociado al evento `OnTimer` es el siguiente:

```
1  procedure TfrmMain.Timer1Timer(Sender: TObject);
2  begin
3    with dmCachedUpdates.ProductcategoryTable, ↙
       ↳ btnApplyUpdates do
4    begin
5      if UpdatesPending then
6      begin
7        Text := 'ApplyUpdates␣(' + IntToStr(ChangeCount) + ↙
           ↳ ')';
8        Enabled := True;
9        btnCancelUpdates.Enabled := True;
10     end
11     else
12     begin
13       Text := 'ApplyUpdates';
14       Enabled := False;
15       btnCancelUpdates.Enabled := False;
16     end;
17   end;
18 end;
```

Listado 11.7 Actualización periódica de los botones

La mejor forma de comprobar el funcionamiento de la actualización por lotes consiste en ejecutar dos o más instancias de este programa. Algunos de los escenarios de experimentación podrían ser:

- Modificación, inserción y eliminación de filas en la primera instancia. Uso del `TCheckBox` para mostrar los cambios. Refresco de los datos en la segunda instancia y los cambios no son visibles. Aplicación de los cambios en la primera y refresco en la segunda, ahora los cambios ya están en la

base de datos. Tras aplicar los cambios el botón de filtrado no muestra nada pendiente.

- Modificación, inserción y eliminación de filas en una de las instancias. Uso del botón de cancelación de los cambios. Se comprueba cómo en la cuadrícula los datos vuelven a su estado original.
- Modificación de datos de una misma fila en ambas instancias y aplicación de los cambios. La operación fallará en la segunda instancia que intente el cambio (véase la Figura 11.3).

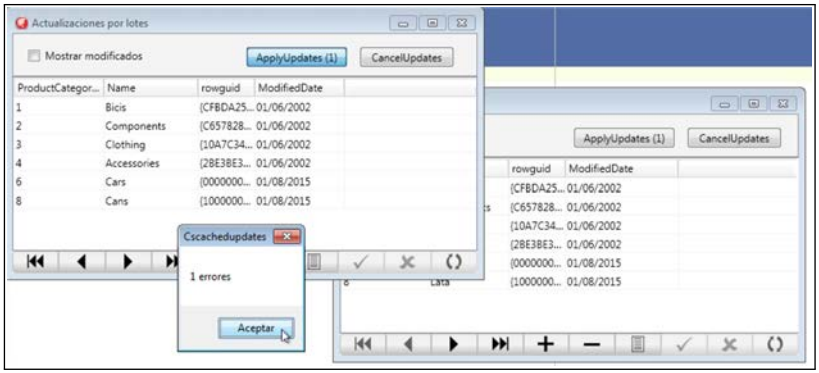


Figura 11.3 LA MODIFICACIÓN SIMULTÁNEA GENERA UN ERROR

NOTA

Este ejercicio puede completarse agregando al módulo de datos un componente `TFDMonitorRemoteClientLink` *y el uso de la herramienta de supervisión a fin de comprobar cómo se produce la comunicación con el servidor.*

Dado que tras invocar a `ApplyUpdates` *no se llama a* `CommitUpdates` *el registro de cambios no se vacía, por lo que el programa seguirá indicando que hay cambios pendientes de enviar al servidor. Basta con agregar una llamada a* `CommitUpdates` *para completar la funcionalidad del ejercicio.*

11.2.4 Gestión de conflictos

Una parte esencial de la actualización por lotes estriba en gestionar adecuadamente los conflictos que pudieran surgir al transferir los cambios al servidor de datos. El método `ApplyUpdates` acepta como parámetro un entero especificando el número máximo de errores que se permitirá antes de interrumpir el envío de comandos de actualización al servidor. La información relativa a los errores quedará almacenada junto con el estado de cada fila afectada. El valor devuelto por dicho método indica el número de errores encontrados.

El método `Reconcile` provocará que el componente `TFDQuery` (u otro conjunto de datos) genere un evento `OnReconcileError` por cada uno de los conflictos que se encuentren. El método asociado a dicho evento recibirá los siguientes parámetros:

- `DataSet`: Un objeto de tipo `TFDDataSet` conteniendo la fila de datos afectada por el error. Podemos usar el método `FieldByName` para obtener una referencia a la columna que nos interese. Las propiedades `Value`, `OldValue`, `CurValue` y `NewValue` de la columna permiten obtener el valor actual, valor original y nuevo valor. Podemos usar esta última propiedad para modificar dicho valor en caso de que fuese necesario.
- `E`: Un objeto de tipo `EFDException` con datos relativos al error, incluyendo su código (`FDCode`), objeto afectado (`FDObjName`), mensaje de error (`Message`), etc.
- `UpdateKind`: Uno de los valores de la enumeración `TFDDatSRowState` que servirá para conocer la operación que falló. Los valores habituales son `rsInserted`, `rsDeleted`, `rsModified` y `rsUnchanged` indicadores de que la fila ha sido añadida, eliminada, modificada o no tiene cambio alguno pendiente.
- `Action`: Este es un parámetro de salida, no de entrada, que servirá para indicar al conjunto de datos qué debe hacer para solventar el error. Las acciones disponibles son las siguientes:
  - `raSkip`: Ignora el error que afecta a la fila actual y continúa con el proceso de resolución de conflictos.
  - `raAbort`: Cancela todo el proceso de resolución de conflictos.

- `raMerge`: Se combinan los cambios efectuados por otros usuarios con los de nuestra instancia de la fila y se transfiere el nuevo cambio.
- `raCorrect`: Los valores que hemos asignado serán los que se transfieran al servidor, sobrescribiendo los de otros usuarios. Dentro del propio evento podemos usar la propiedad `NewValue` de las columnas para cambiar los valores a usar.
- `raCancel`: Se cancelan los cambios locales, devolviendo la fila a sus valores originales.
- `raRefresh`: Se cancelan los cambios locales, recuperando los nuevos valores de la fila del servidor.

Los cuatro últimos valores de `Action` borrarán el indicador de error de la fila afectada, no así los dos primeros. De ello dependerá que el método `Reconcile` devuelva finalmente el valor `True`, indicando que ha sido posible solventar todos los conflictos existentes, o `False`, comunicando que aún hay algún error pendiente de solucionar.

Gestionando adecuadamente el evento `OnReconcileError` podemos exponer al usuario el problema que se ha encontrado al enviar los cambios al servidor, consultándole la operación que desea llevar a cabo.

NOTA

Una alternativa al uso del evento `OnReconcileError` consiste en usar el evento `OnUpdateError` del conjunto de datos. Este se produce antes de anotar en el registro local de cambios (propiedad `Delta`) el resultado de la operación, por lo que permite solventar los problemas en una fase previa.

Partiendo del proyecto desarrollado anteriormente, veamos qué cambios tendríamos que introducir para implementar un mecanismo básico de resolución de conflictos. El primer paso será modificar el código asociado al botón que llamaba al método `ApplyUpdates`, de forma que también invoque al método `Reconcile` y, solo si este devuelve `True`, borre el registro local de cambios mediante el método `CommitUpdates`. El método quedaría como se muestra a continuación:

```
procedure TfrmMain.btnApplyUpdatesClick(Sender: TObject);
begin
  with dmReconcile.ProductcategoryTable do
  begin
    ApplyUpdates;
    if Reconcile then
      CommitUpdates;
  end;
end;
```

Listado 11.8 Nuevo código del botón de aplicación de cambios

A continuación abriremos el módulo de datos, seleccionaremos el componente `TFDQuery`, buscaremos el evento `OnReconcileError` en el Inspector de objetos y haremos doble clic sobre él para abrir el método correspondiente. El código de este se limitará a llamar a un método de un formulario que aún no hemos definido, facilitándole los parámetros necesarios para gestionar el conflicto. Dicho método devolverá como resultado la acción a ejecutar, valor que asignaremos al parámetro `Action` tal y como se muestra a continuación:

```
procedure TdmReconcile.ProductcategoryTableReconcileError(↙
    ↳ DataSet: TFDDataSet;
  E: EFDException; UpdateKind: TFDDatSRowState;
  var Action: TFDDAptReconcileAction);
begin
  Action := frmReconcile.ReconcileForm(DataSet, E, ↙
      ↳ UpdateKind);
end;
```

Listado 11.9 El módulo de datos delega la gestión de los conflictos en un formulario externo

El siguiente paso será agregar al proyecto un formulario adicional. Encontrándonos en el módulo de dato, utilizaremos la opción **FILE—USE UNIT** para agregar una referencia a ese nuevo formulario (véase la Figura 11.4).

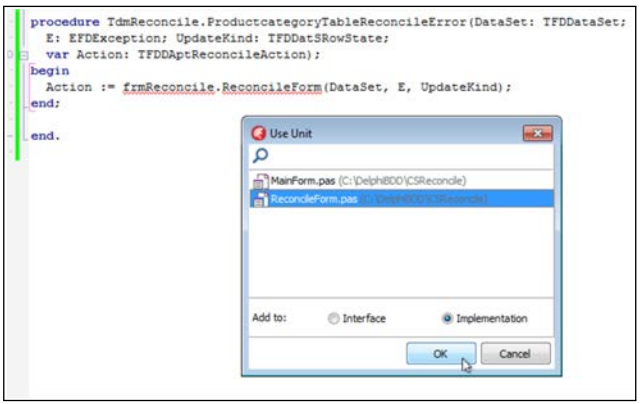


Figura 11.4 ENLAZAMOS EL MÓDULO DE DATOS CON EL NUEVO FORMULARIO

EJEMPLO 11.3 `CSReconcile.dproj`

Puedes encontrar este proyecto completo en la carpeta `CSReconcile`. Deberás aplicar los mismos ajustes que en el proyecto previo.

Insertaremos en el nuevo formulario un conjunto de controles `TLabel` y `TEdit` que nos servirán para mostrar información sobre el error que se ha producido, así como los valores originales, actuales y nuevos de dos de las columnas del conjunto de datos. También añadiremos un `TComboBox` y un `TButton`. En la Figura 11.5 puede apreciarse la distribución y texto de las etiquetas estáticas, así como el nombre de las que cambiarán su contenido (en negrita) y el nombre de cada uno de los `TEdit`. Daremos el valor `True` a la propiedad `ReadOnly` de los seis `TEdit`, de forma que sirvan únicamente para mostrar los valores pero no su modificación.

El control `TComboBox` permitirá al usuario elegir la acción que desea aplicar para solventar el conflicto. Abrimos el editor asociado a dicho control (véase la Figura 11.6) y creamos un elemento para cada acción posible. Es importante respetar el orden y asignar a las respectivas propiedades `Tag` los valores `0` a `5`.

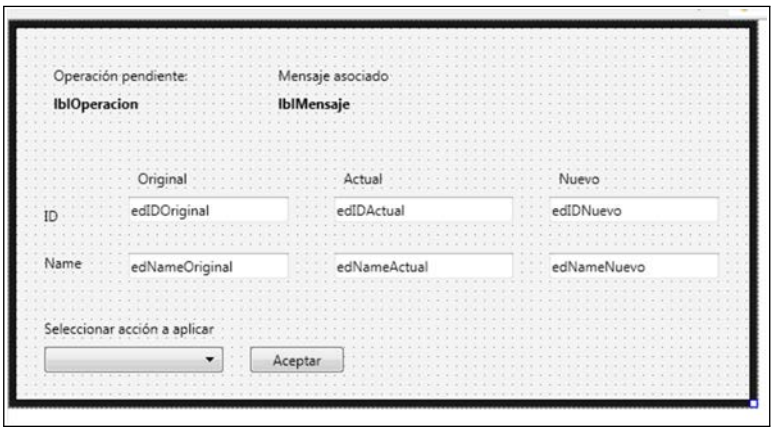


Figura 11.5 ELEMENTOS DEL FORMULARIO DE RESOLUCIÓN DE CONFLICTOS

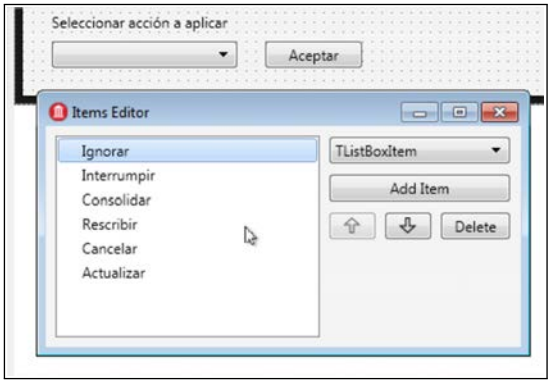


Figura 11.6 LISTA DE ACCIONES EN EL `TComboBox`

Este formulario ha implementar un método público `ReconcileForm`. Este es el método al que se llamaba desde el módulo de datos. Tras agregarlo a la definición de clase procederemos a implementar su funcionalidad que, como se aprecia en el siguiente listado, consta de tres pasos básicos: visualización de los datos relativos al conflicto, mediante la asignación de los parámetros de entrada

a la propiedad `Text` de los controles; apertura del formulario para que el usuario pueda verlos y decidir, usando el método `ShowModal`, y devolución del valor seleccionado en la lista de acciones:

```
  public
    function ReconcileForm(DataSet: TFDDataSet;
      E: EFDException; UpdateKind: TFDDatSRowState): ↙
          ↳ TFDDAptReconcileAction;
  end;

...

implementation

function TfrmReconcile.ReconcileForm;
begin
  with lblOperacion do
  case UpdateKind of
    rsInserted: Text := 'Inserción';
    rsDeleted: Text := 'Eliminación';
    rsModified: Text := 'Actualización';
    rsUnchanged: Text := 'Ninguna';
  end;

  lblMensaje.Text := E.Message;

  with DataSet do
  begin
    edIDOriginal.Text := FieldByName('ProductCategoryID').↙
        ↳ OldValue;
    edIDActual.Text := FieldByName('ProductCategoryID').↙
        ↳ CurValue;
    edIDNuevo.Text := FieldByName('ProductCategoryID').Value↙
        ↳ ;
    edNameOriginal.Text := FieldByName('Name').OldValue;
    edNameActual.Text := FieldByName('Name').CurValue;
    edNameNuevo.Text := FieldByName('Name').Value;
  end;
```

```
31
32    ShowModal;
33
34    result := TFDDAptReconcileAction(cbAction.Selected.Tag);
35  end;
```

Listado 11.10 Implementación del método de gestión de conflictos

Tan solo resta la implementación del método asociado al evento `OnClick` del botón. Este se limitará a cerrar el formulario:

```
1  procedure TfrmReconcile.Button1Click(Sender: TObject);
2  begin
3    Close;
4  end;
```

Listado 11.11 El botón cerrará el formulario

Para comprobar esta nueva funcionalidad tendremos que ejecutar al menos dos instancias del programa y modificar la misma fila en ambos, generando un conflicto que desencadenará la apertura del nuevo formulario, como se aprecia en la Figura 11.7.

11.2.5 Actualización por lotes centralizada

En los ejercicios previos hemos utiliza únicamente un conjunto de datos, por lo que no tenía sentido recurrir al mecanismo de centralización de la actualización por lotes que ofrece FireDAC. En realidad esta tampoco es imprescindible si se opera con varios conjuntos de datos independientes entre sí, ya que cada uno es capaz de mantener su registro local de cambios y enviar los comandos adecuados al servidor cuando se demande.

La situación se hará más compleja si en la aplicación existen relaciones maestro/detalle. En estos casos es necesario respetar el orden cronológico de las operaciones, ya que las entradas del conjunto de datos de detalle hará referencia al maestro, por lo que cada fila de este ha de ser gestionada antes que las filas de detalle que dependen de ella. Se requiere un mecanismo de actualización por lotes centralizado. Este está representado por el componente `TFDSchemaAdapter`.

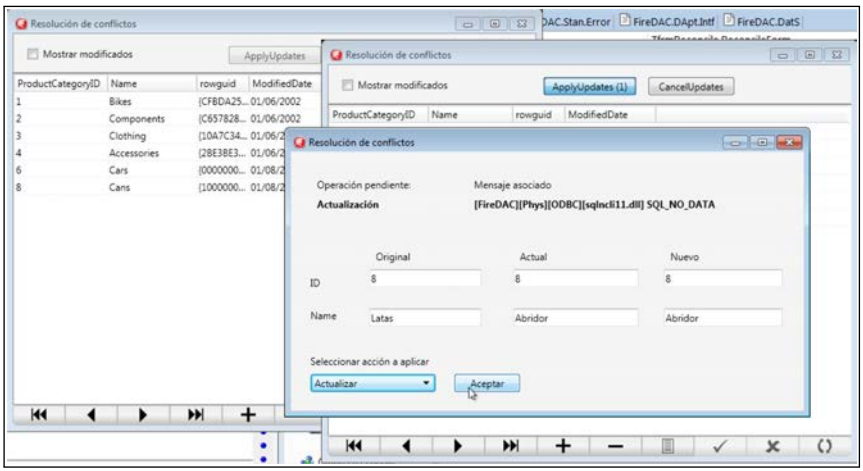


Figura 11.7 EL PROGRAMA INTENTADO SOLVENTAR UN CONFLICTO

Los componentes derivados de la clase `TFDDataSet`, entre los que se cuentan `TFDQuery` y `TFDTable`, cuentan con la propiedad `SchemaAdapter`. Su finalidad es vincular el conjunto de datos con el `TFDSchemaAdapter`, de forma que se delegue en este último la gestión del registro local de cambios. Para gestionar los cambios usaríamos los métodos y eventos de dicho componente, por ejemplo llamando a `ApplyUpdates` o respondiendo al evento `OnReconcileError`, todo ello de forma centralizada, en lugar de actuar sobre cada conjunto de datos por separado.

> **NOTA**
>
> *Al usar la actualización por lotes centralizada con relaciones de tipo maestro/detalle es necesario dar el valor `True` al campo `DetailCascade` de la propiedad `FetchOptions` de los conjuntos de datos de detalle. De esta forma los cambios introducidos en el conjunto maestro se propagarán al conjunto de detalle.*

11.3 Resumen

Al finalizar este capítulo hemos aprendido a trabajar con el modo desconectado de FireDAC, gracias al cual cada cliente puede operar de forma local sin necesidad de mantener una conexión permanente con el servidor de datos. Este modo se complementa perfectamente con la actualización de datos por lotes, una característica que también es útil por sí misma al reducir el número de operaciones de transferencia de datos y facilitar acciones habituales como la de deshacer.

En el siguiente capítulo, último de esta parte, centraremos nuestra atención en algunos detalles concretos que afectan a dos de los RDBMS más usados cuando se desarrollan aplicaciones con Delphi: InterBase y Microsoft SQL Server.

Capítulo 12

INTERBASE

En los capítulos previos de esta parte del libro se han descrito procedimientos genéricos de trabajo con FireDAC en un entorno cliente/servidor. Aunque en los ejercicios se ha utilizado una base de datos SQL Server, en la práctica podríamos usar cualquier otro origen de datos simplemente cambiando la configuración de conexión, el resto de los elementos apenas necesitarían ajustes.

Entre los RDBMS con los que es posible trabajar con FireDAC InterBase ocupa un lugar algo especial. Es el servidor de datos con el que Delphi tiene mayor afinidad, al proceder ambos productos de la misma empresa. InterBase es, además, uno de los pocos servidores de datos preparados para operar en en-

tornos que van desde las aplicaciones móviles, con el servidor embebido, hasta las aplicaciones distribuidas y cliente/servidor, con InterBase operando en un servidor Windows o GNU/Linux. Esta disponibilidad facilita el desarrollo de soluciones software multiplataforma y la transición entre plataformas, al residir los datos siempre en un mismo formato y ser gestionados fundamentalmente por el mismo software.

El objetivo de este capítulo es el de ofrecer detalles adicionales sobre la configuración y el trabajo con un servidor de datos InterBase desde aplicaciones Delphi con componentes FireDAC.

12.1 Versiones de InterBase

InterBase es un RDBMS disponible en múltiples versiones, todas ellas compartiendo un mismo núcleo de características, aunque con distintas limitaciones, y utilizando un mismo formato de almacenamiento. Las diferencias esenciales entre las versiones existentes son las destacadas a continuación:

- **Lite y ToGo**: Son las dos versiones diseñadas para su uso embebido, sin precisar instalación. Están disponibles para Windows, OS X, Android e iOS y las principales diferencias entre ambas ya fueron enumeradas en la Sección 2.2.1 (página 77).
- **Desktop**: Es una versión de InterBase diseñada específicamente para su uso como base de datos local en Windows. Solo permite conexiones de un usuario desde el mismo equipo en que se ejecuta InterBase. Por lo demás cuenta con las mismas características básicas de la versión Server.
- **Server**: Esta es la versión adecuada para desarrollos cliente/servidor y distribuidos. Puede ser instalada en Windows o GNU/Linux, ya sea en versiones de 32 o de 64 bits. Puede aprovechar configuraciones multiprocesador y permite el acceso simultáneo a un número ilimitado de usuarios, ya sean locales o remotos (con conexión a través de TCP/IP).
- **Developer**: Se trata de la versión de desarrollo de InterBase Server, con la misma funcionalidad de dicha edición pero sin licencia para su instalación en entornos de producción. Sería la que utilizaríamos con Delphi para desarrollar y comprobar nuestros proyectos.

Como usuarios de Delphi, lo habitual es que utilicemos la versión Developer durante el desarrollo del proyecto a fin de prepararlo según la configuración que tendría en el entorno de explotación, en el que se instalaría finalmente la versión Server con su correspondiente licencia. Asumiendo que trabajaremos con esta última, instalada en un servidor Windows o GNU/Linux, veamos ahora cuál sería la configuración del lado cliente.

12.2 Configuración del cliente

Asumiendo que tengamos InterBase Server (o Developer) funcionando en un equipo remoto como servidor de datos, para poder conectar con él nuestra aplicación dependerá de un elemento externo: el software cliente de InterBase para la plataforma en que vaya a ejecutarse el programa.

El componente `TFDConnection`, a través del proveedor representado por el componente `TFDPhysIBDriverLink`, se comunicará con el software cliente enviándole los comandos a ejecutar. Será el software cliente el que, a la postre, contacte con el servidor InterBase mediante TCP/IP y efectúe la transferencia del comando y de los resultados obtenidos, entregando estos de vuelta a la aplicación.

Dependiendo del sistema en que vaya a ejecutarse la aplicación el software cliente a distribuir conjuntamente con esta sería el indicado a continuación:

- **Windows**: Para aplicaciones Windows de 32 bits el archivo a redistribuir es `gds32.dll`, mientras que para 64 bits sería `ibclient64.dll`.

- **OS X**: El archivo a redistribuir es `libgds.dylib`, equivalente a la biblioteca `gds32.dll` de Windows.

Para incluir la biblioteca adecuada con la aplicación, de forma que sea distribuida junto a esta, podemos usar la opción **PROJECT—DEPLOYMENT**, concretamente el cuadro de diálogo **FEATURED FILES** (véase la Figura 12.1), marcando la plataforma que nos interese.

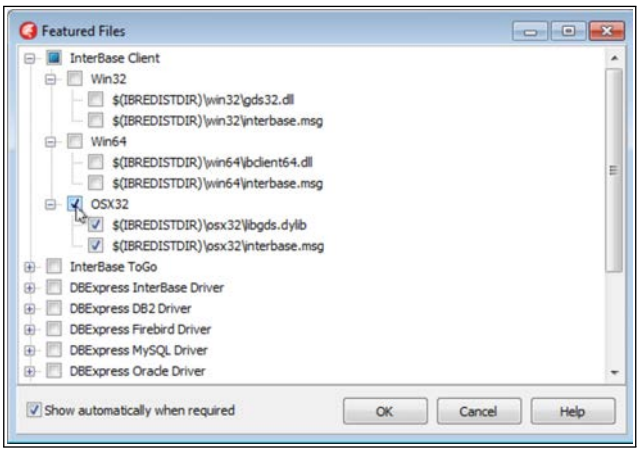


Figura 12.1 DISTRIBUCIÓN DEL CLIENTE INTERBASE CON LA APLICACIÓN

> **NOTA**
>
> *La biblioteca con el software cliente puede alojarse en la misma carpeta en la que resida el ejecutable de nuestra aplicación o en una carpeta de sistema. Si se usa cualquier otra localización deberemos facilitar la ruta mediante la propiedad* `VendorLib` *del componente* `TFDPhysIBDriverLink` *o la entrada homónima del archivo de configuración* `FDDrivers.ini` *de FireDAC. Este último debe alojarse en la carpeta donde esté el ejecutable.*

Entre las propiedades del componente `TFDPhysIBDriverLink`, cuyos valores por defecto suelen ser apropiados, podemos usar las dos siguientes para ajustar la configuración del controlador[1]:

[1]Cualquier cambio en la configuración del controlador deberá efectuarse antes de abrir la conexión con la base de datos para que tenga efecto.

- `VendorHome`: Ha de contener la ruta completa de la carpeta en la que está instalado el software cliente.
- `VendorLib`: Contendrá el nombre de la biblioteca con el software cliente. Será uno de los archivos antes enumerados.

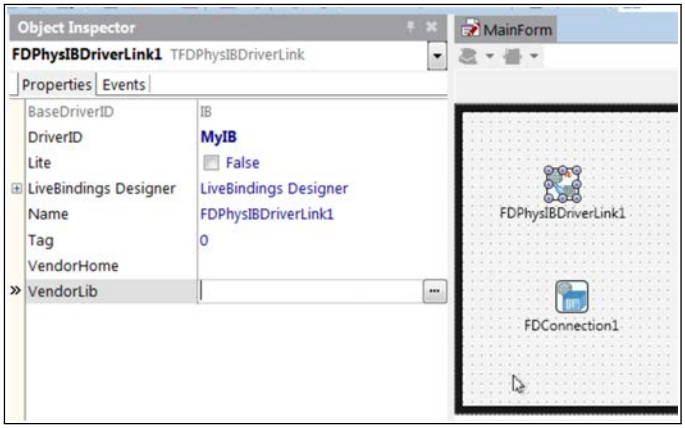


Figura 12.2 CONFIGURACIÓN DEL CONTROLADOR FIREDAC PARA INTERBASE

12.3 Configuración de la conexión

En los ejercicios en los que usábamos una base de datos InterBase, en los capítulos de la parte anterior, el archivo con la información residía en el mismo equipo donde se ejecutaba la aplicación, por lo que bastaba con facilitar la ruta y el nombre de dicho archivo. Al trabajar en una configuración cliente/servidor, sin embargo, tendremos que facilitar información adicional para configurar la conexión.

Otros parámetros que pueden especificarse en la conexión a InterBase afectan al modo de autenticación, incluyendo las credenciales si son necesarias, la codificación de caracteres a emplear, etc. En los siguientes apartados se detallan los más importantes y se facilitan algunos ejemplos de conexión.

12.3.1 Protocolo, servidor y puerto

En la propiedad `Params` del componente `TFDConnection` encontramos tres campos que establecen el protocolo que se utilizará para comunicarse con el servidor de datos, el equipo que actúa como tal y el puerto de comunicación en el que InterBase está a la escucha. Dichos campos y su contenido son los descritos a continuación:

- `Protocol`: InterBase, tanto el servidor como el software cliente asociado, está preparado para operar en distintos tipos de redes, usando para ello protocolos específicos de cada una de ellas. Con esta propiedad elegiremos el protocolo a utilizar. Si nos encontrásemos en una configuración cliente/servidor en una red Windows, podríamos usar el protocolo NetBEUI para comunicar la aplicación con el RDBMS. También se contempla el proto-colo SPX. No obstante, lo más habitual es utilizar el protocolo TCP/IP ya que este facilita la comunicación entre redes compuestas de máquinas con distintos sistemas.

- `Server`: Dependiendo del protocolo seleccionado en la propiedad previa, en esta se identificará el servidor en el que está ejecutándose InterBase Server de una forma u otra. Lo habitual es utilizar el nombre asignado a la máquina, por ejemplo con el protocolo NetBEUI, o bien una dirección IP si se opta por TCP/IP. En este último caso también podría facilitarse un nombre de servidor resoluble mediante DNS, del tipo `rdbms.empresa.com`, suponiendo que tuviésemos un dominio `empresa.com` con un equipo llamado `rdbms`.

- `Port`: Por defecto InterBase utiliza el puerto 3050 para comunicarse. Este será el puerto en el que esté a la escucha el servidor y al que el software cliente enviará sus solicitudes. Con la propiedad `Port` podemos cambiar dicho puerto por cualquier otro.

- `InstanceName`: En caso de que la instancia[2] de InterBase con la que queremos conectar no sea la instancia por defecto, mediante esta propiedad facilitaremos el nombre que tiene asignado.

[2]InterBase contempla la ejecución de múltiples instancias del servidor de datos en un mismo servidor, cada una de ellas con un nombre diferente.

NOTA

Con estas propiedades estaríamos estableciendo los parámetros de comunicación del software cliente de InterBase. Lógicamente habría que ajustar la configuración del servidor con esos mismos parámetros, de lo contrario no sería posible la comunicación. Podemos obtener información sobre los protocolos activos, el puerto en el que está a la escucha y la licencia de InterBase Server abriendo la ventana de propiedades de la utilidad InterBase Server Manager (véase la Figura 12.3).

Figura 12.3 INFORMACIÓN SOBRE INTERBASE SERVER

12.3.2 Autenticación

Dependiendo de que InterBase Server esté ejecutándose en Windows o en Linux podremos optar entre distintos tipos de autenticación. En el primer sistema puede

elegirse entre la autenticación integrada de Windows, dando el valor `True` a la propiedad `OSAuthent`, o bien la autenticación propia del RDBMS, dejándola a `False` que es su valor por defecto. En GNU/Linux únicamente tendremos la segunda opción.

Si se opta por la autenticación del RDBMS, que es la opción por defecto, será preciso facilitar las credenciales apropiadas en las propiedades `UserName` y `Password`. Ya sabemos que por defecto InterBase utiliza `sysdba` como usuario y `masterkey` como contraseña por defecto, pero en un servidor en explotación dicha cuenta debería estar desactivada. En su lugar utilizaríamos las credenciales que nos facilitase el administrador de la base de datos.

12.3.3 Identificación de la base de datos

La base de datos a la que quiere accederse ha de indicarse en la propiedad `Database`, como es habitual. Al trabajar en un entorno cliente/servidor, sin embargo, la ruta y el nombre del archivo, que era lo que facilitábamos en ejercicios de capítulos previos, puede complementarse con otros datos. De hecho, parte de la información que entregábamos en propiedades como `Server` y `Port` puede introducirse directamente en la propiedad `Database`.

El formato general para la propiedad `Database` es el siguiente:

```
nombre-servidor/puerto-comunicacion:ruta/archivo-bdd
```

El nombre del servidor, que también puede una dirección IP, por defecto será el asignado a la propiedad `Server`. Análogamente, el puerto de comunicación por defecto sería el indicado por la propiedad `Port`. El carácter `:` actúa como separador de la identificación del servidor y la de la base de datos. Con una configuración como la siguiente:

```
rdbms.empresa.com/3050:D:/archivo.ib
```

Accederíamos a la base de datos `archivo.ib`, alojada en un servidor llamado `rdbms.empresa.com`. Conseguiríamos el mismo resultado asignando a `Database` solamente la ruta `D:/archivo.ib` y almacenando el nombre del servidor y puerto en las citadas propiedades `Server` y `Port`.

> **NOTA**
>
> *Puedes obtener más información sobre la configuración de los parámetros de conexión en la guía de operaciones de InterBase XE7, disponible en* `http://docs.embarcadero.com/products/interbase/IBXE7/OpGuide.pdf`.

12.3.4 Otros parámetros

Además de los enumerados en los apartados previos, la propiedad `Params` acepta varios campos más que afectan al funcionamiento de InterBase al establecer una conexión con la base de datos. Las más destacables son las siguientes:

- `OpenMode`: Al conectar con la base de datos por defecto se pedirá al servidor que la abra para trabajar con ella, generándose una excepción en caso de no existir. Las otras dos alternativas son crear la base de datos en caso de que no exista o crearla siempre, dependiendo del valor que se asigne a esta propiedad.
- `DropDatabase`: Esta propiedad de tipo booleana toma por defecto el valor `false`, de forma que al desconectar de la base de datos el servidor sencillamente la cerrará. Asignándole el valor `true` a esa acción seguiría la eliminación de la base de datos.
- `CharacterSet`: Como ya sabemos, la codificación de la información almacenada en una base de datos ha de estar en concordancia con la esperada por los clientes, ya que de lo contrario los datos obtenidos podrían resultar totalmente ilegibles. Mediante esta propiedad estableceremos la configuración en el cliente, como se aprecia en la Figura 12.4.

Una vez establecidos todos los parámetros llegará el momento de activar la conexión. Esta puede demorarse en el tiempo, dependiendo de parámetros como el ancho de banda y la carga del servidor de datos. A partir de ese punto usaríamos las técnicas descritas en los capítulos previos para operar sobre los datos, como haríamos con cualquier otro RDBMS, incluyendo aspectos como las transacciones, bloqueos, actualizaciones por lotes, etc.

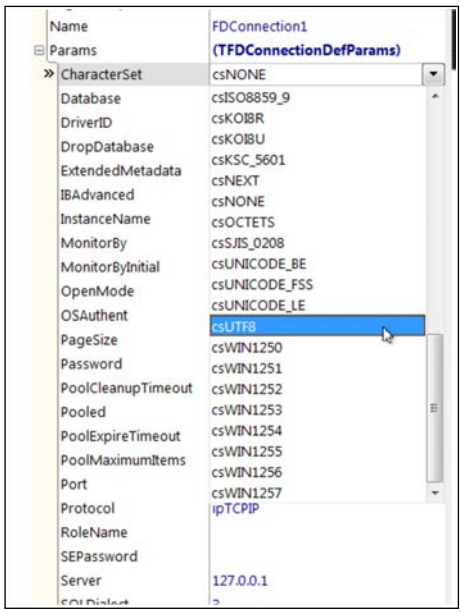


Figura 12.4 CONFIGURACIÓN DE LA CODIFICACIÓN A EMPLEAR

12.4 Componentes FireDAC Services

InterBase se distribuye con un conjunto de pequeñas utilidades, accesibles desde la línea de comandos en Windows y la consola de la *shell* en GNU/Linux, diseñadas para facilitar distintas tareas de mantenimiento: copia de seguridad y restauración de bases de datos, recuperación de transacciones, gestión de los usuarios que tienen acceso a la misma, etc. Cada uno de estos programas: `gbak`, `gsec`, `gfix`, etc., acepta un conjunto determinado de argumentos con los que se indica la acción a llevar a cabo, la base de datos sobre la que se operará y otros parámetros dependientes de cada función.

Si tenemos instalada la aplicación IBConsole (en Windows se instalará junto con el servidor) podemos llevar a cabo todas esas operaciones cómodamente desde la interfaz gráfica de usuario, tal y como se aprecia en la Figura 12.5. El

menú contextual de cada base de datos da paso a todas las opciones de mantenimiento, agrupadas según la funcionalidad.

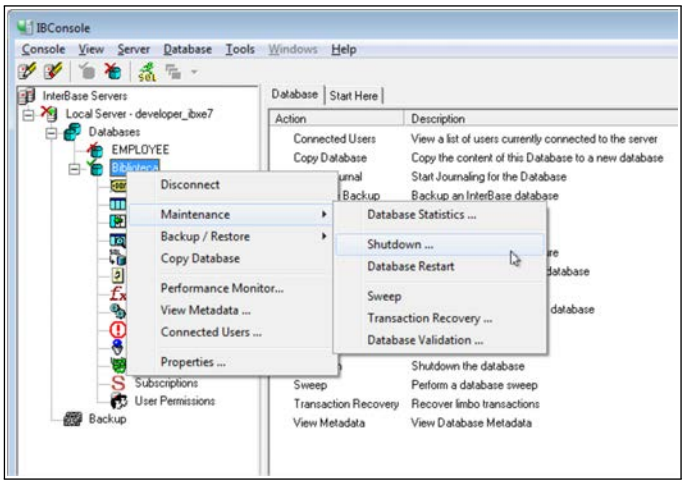


Figura 12.5 OPCIONES DE MANTENIMIENTO

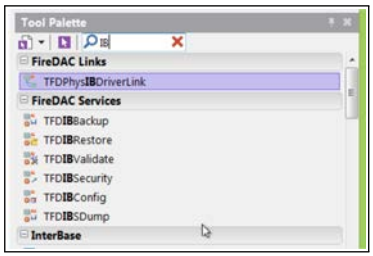


Figura 12.6 COMPONENTES DE SERVICIO PARA INTERBASE

Las mencionadas utilidades, así como la interfaz de IBConsole, permiten conectar con un servidor remoto de InterBase y actuar sobre él, de forma que gran parte del mantenimiento puede efectuarse de manera remota. Estas tareas pueden ser también incorporadas en nuestras aplicaciones, gracias a los componentes de servicio específicos para InterBase con que cuenta FireDAC. Encontramos dichos componentes en la página **FIREDAC SERVICES** (véase la Figura

12.6) de la Paleta de componentes. La denominación de cada uno es bastante descriptiva y deja entrever la funcionalidad que ofrece. En los apartados de esta sección se describe brevemente la metodología de uso de estos componentes y se enumera la finalidad de cada uno de ellos.

> **NOTA**
>
> *En general, la funcionalidad ofrecida por los componentes descritos a continuación es aplicable también a Firebird, el RDBMS open source basado en una antigua versión de InterBase.*

12.4.1 Metodología general de uso

La mayor parte de los componentes de servicio de InterBase actúan sobre una base de datos concreta, utilizando para el controlador (representado por el componente `TFDPhysIBDriverLink`) que habremos incluido en el modulo de datos o formulario. Por ello todos estos componentes cuentan con una propiedad, llamada `DriverLink`, cuya finalidad es enlazarlos con dicho controlador. No hay más que desplegar la lista asociada a la propiedad y elegir el componente `TFDPhysIBDriverLink`.

Establecido el vínculo entre componente de servicio y controlador, generalmente tendremos que configurar las siguientes propiedades del primero:

- `UserName` y `Password`: Nombre y contraseña de un usuario registrado en el servidor y con las credenciales necesarias para poder operar sobre él.
- `Host` e `InstanceName`: Identificación del equipo en el que está ejecutándose InterBase Server y, en caso necesario, la instancia del RDBMS sobre la que se quiere trabajar.
- `Database`: Nombre de la base de datos sobre la que recaerán las operaciones demandadas por el componente de servicio.

Una vez identificado el servidor, instancia, base de datos y usuario, el procedimiento habitual será invocar a los métodos ofrecidos por el componente de servicio para realizar la operación deseada.

12.4.2 Funcionalidad de los componentes de servicio

En la citada página de la Paleta de componentes encontraremos media docena de componentes de servicios para InterBase. La funcionalidad básica ofrecida por cada uno de ellos es la descrita a continuación:

- `TFDIBConfig`: Permite establecer distintos parámetros de configuración de una base de datos, como el modo de acceso, tamaño de la caché, frecuencia de recolección de basura, etc. También facilita métodos para poner una base de datos en modo mantenimiento (*offline*), de forma que no se aceptarían conexiones a la misma, o en modo normal (*online*).
- `TFDIBValidate`: Los métodos que ofrece este componente permiten analizar el estado de una base de datos, así como repararla en caso de que fuese necesario o ejecutar el proceso de limpieza para reducir su tamaño y optimizar el rendimiento.
- `TFDIBBackup`: Como es fácil deducir por su nombre, la finalidad de este componente es realizar una copia de seguridad del contenido de una base de datos. Para ello se leen las filas de las tablas y se escriben en un archivo externo en un formato específico para la copia, no en el formato propio de la base de datos.
- `TFDIBDump`: La finalidad de este componente es similar a la del anterior, pero la copia del contenido de la base de datos se efectúa al nivel de página y preservando el mismo formato original, existiendo además la posibilidad de efectuar copias incrementales.
- `TFDIBRestore`: Ofrece la funcionalidad complementaria a los dos componentes anteriores, restaurando el contenido de una base de datos a partir de un archivo de copia de seguridad, sin que importe con qué componente fue generado.
- `TFDIBSecurity`: Es el componente que facilita las operaciones de gestión de usuarios, incluyendo la adición y eliminación de usuarios, cambio de la contraseña asociada, obtención de los usuarios existentes, etc.

Dado que con las herramientas citadas anteriormente, como IBConsole, es posible realizar todas estas tareas de forma cómoda sobre cualquier base de datos

InterBase, la utilidad de estos componentes la encontraremos fundamentalmente cuando deseemos incorporar en una aplicación alguna de estas funcionalidades pero limitada a la base de datos sobre la que se trabaja. Serviría, por ejemplo, para ofrecer al usuario opciones de análisis y reparación de la base de datos, realización de copias de seguridad y funciones similares.

12.5 Resumen

En este capítulo se han descrito las características más destacables de InterBase como servidor de datos en un entorno cliente/servidor, así como los aspectos específicos a la hora trabajar con este RDBMS usando Delphi y FireDAC. Asimismo se han introducido los componentes de servicio que permiten realizar tareas de mantenimiento sobre el servidor.

Los capítulos de la siguiente parte del libro abordan un contexto algo más complejo que el tratado hasta ahora, con aplicaciones diseñadas según una arquitectura distribuida en lugar de cliente/servidor. Básicamente es la primera es una extensión de esta última, en la que el cliente se desacopla en dos partes dando lugar a aplicaciones formadas por tres capas.

PARTE 3

ACCESO A DATOS EN APLICACIONES DISTRIBUIDAS

En esta tercera parte del libro nos ocuparemos de diferentes aspectos relacionados con el desarrollo de aplicaciones distribuidas, si bien algunas de las técnicas que se describirán son aplicables asimismo en configuraciones cliente/servidor. La arquitectura distribuida, habitualmente formada por tres capas: una interfaz de usuario, un servidor de aplicaciones y un servidor de datos. La comunicación entre las distintas capas puede ser local o a través de Internet.

Delphi ofrece los componentes necesarios para implementar servidores que ofrecen tanto operaciones como conjuntos de datos, accesibles remotamente mediante TCP/IP y HTTP, así como los elementos apropiados para construir clientes que accedan a los mismos. En realidad estos últimos pueden utilizarse para consumir cualquier servicio de tipo REST (*Representational State Transfer*) al que tengamos acceso.

Comenzaremos introduciendo todos los conceptos necesarios, relativos a la tecnología DataSnap para aplicaciones distribuidas y la naturaleza de los servicios REST. A continuación aprenderemos a desarrollar módulos de servidor DataSnap, ofreciendo servicios que aprenderemos a consumir desde aplicaciones cliente desarrolladas con Delphi. Finalmente se introducirán los EMS (*Enterprise Mobility Services*), una capa de servicios disponible en las últimas versiones de Delphi.

Introducción al desarrollo distribuido con Delphi **375**

Servicios DataSnap **393**

Servicios REST **425**

Introducción a EMS **469**

Capítulo 13

INTRODUCCIÓN AL DESARROLLO DISTRIBUIDO CON DELPHI

En la actualidad el desarrollo de aplicaciones distribuidas con Delphi está íntimamente ligado a DataSnap, la tecnología introducida en Delphi 6 y heredera de MIDAS. En la Sección 1.4.5 (página 52) se introdujeron los aspectos fundamentales de DataSnap, así como la arquitectura de una aplicación basada en este *framework*.

Aunque DataSnap, por compatibilidad hacia atrás, sigue considerando la posibilidad de comunicar las distintas partes de una aplicación distribuida mediante DCOM, para nuevos desarrollos lo habitual es que utilicemos TCP/IP o HTTP. Estos protocolos son estándares abiertos y ampliamente aceptados, per-

miten operar fácilmente a través de Internet y contemplan distintos mecanismos de seguridad. La comunicación directa mediante TCP/IP suele usarse cuando cliente y servidor están implementados en Delphi. Mediante HTTP, y siguiendo la filosofía REST junto con JSON para la representación de datos, se facilita el desarrollo de servidores accesibles desde clientes implementados con otras herramientas y lenguajes de programación.

El objeto de este breve capítulo es introducir los conceptos básicos relativos al desarrollo de este tipo de soluciones usando Delphi. En capítulos posteriores entraremos en los aspectos prácticos, implementando servidores DataSnap y clientes que consuman los servicios ofrecidos por los mismos.

13.1 Estructura de una aplicación distribuida

Comencemos analizando con algo más de detalle la estructura de una aplicación distribuida y, sobre todo, cuál sería la funcionalidad a implementar en cada una de sus capas. Tradicionalmente las aplicaciones distribuidas se han construido según una arquitectura en tres capas: el servidor de datos, el servidor de aplicaciones y los clientes. Esta arquitectura puede variar según las necesidades de cada proyecto.

La Figura 13.1 es una representación esquemática de este modelo. En la parte inferior tenemos el servidor de datos. La capa intermedia es el servidor de aplicaciones, desarrollado con DataSnap, que se comunica con el anterior a través de alguno de los mecanismos de acceso a datos integrados en Delphi, preferentemente FireDAC. La capa superior la ocupan las aplicaciones cliente. Estas actúan como interfaz de usuario, facilitando el acceso a los servicios ofrecidos por el servidor de aplicaciones.

Sobre esta arquitectura base es posible introducir cambios. Por ejemplo, un proyecto en el que no se precise acceder a bases de datos prescindiría del servidor de datos, quedando únicamente con las dos capas superiores: servidor de aplicaciones y clientes. Se convertiría así en una aplicación cliente/servidor de otra categoría, en la que el servidor no ofrece datos sino servicios. La capa intermedia podría, en caso necesario, dividirse en dos partes alojadas en dos servidores

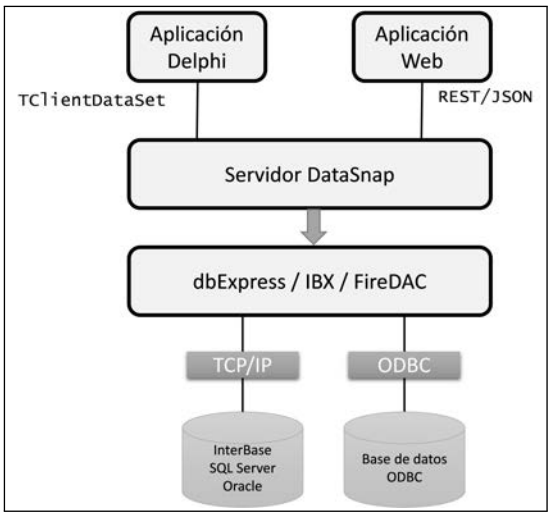


Figura 13.1 ARQUITECTURA DE UNA SOLUCIÓN BASADA EN DATASNAP

distintos, gestionando una de ellas la comunicación con los clientes (gestión de peticiones, autenticación y autorización) y la otra aportando los servicios propiamente dichos. En este caso tendríamos cuatro capas.

Como podemos comprobar, las posibles combinaciones son muchas. Presumiendo que tenemos las tres capas habituales, veamos cuál sería la funcionalidad aportada por cada una de ellas.

13.1.1 Servidor de datos

La capa inferior (atendiendo al esquema de la Figura 13.1) de la aplicación distribuida corresponde al servidor de datos. Este será normalmente un RDBMS como puede ser Oracle, SQL Server o InterBase. La función del servidor de datos será exactamente la misma que una aplicación cliente/servidor, arquitectura de la que nos ocupábamos en la segunda parte del libro.

El servidor de datos será accesible desde el servidor de aplicaciones, por regla general mediante conexiones TCP/IP. Estos dos servidores pueden encontrarse

conectados a la misma red, optimizando el rendimiento de la comunicación entre ambos, o bien ser remotos entre sí y utilizar la infraestructura de Internet para trasladar las peticiones y los datos.

13.1.2 Servidor de aplicaciones

Desde una perspectiva lógica un servidor de aplicaciones ofrece una funcionalidad similar a la de un servidor de datos: es accesible desde el exterior mediante conexiones de red, los clientes envían solicitudes y obtienen una respuesta con el resultado de ejecutarlas. Las dos diferencias fundamentales son:

- Las solicitudes no son sentencias SQL sino llamadas a métodos, incluyendo los parámetros que estos necesiten para realizar su trabajo.
- Las respuestas no tienen necesariamente que ser conjuntos de datos, pudiendo tomar virtualmente cualquier otra forma.

Al igual que un servidor de datos, el servidor de aplicaciones queda a la escucha en un cierto puerto de comunicación y puede atender a múltiples usuarios de forma simultánea, incluyendo opcionalmente funciones de autenticación de usuarios y seguridad de acceso a los servicios ofrecidos.

Las técnicas descritas en la segunda parte de este libro, a la hora de acceder a un servidor de datos en configuración cliente/servidor, son igualmente aplicables en este contexto, si bien el nodo que accede al servidor de datos en este caso no es una aplicación cliente sino los componentes del servidor de aplicaciones.

> **NOTA**
>
> *Con DataSnap el servidor de aplicaciones puede ser una aplicación ejecutable, un servicio Windows o un módulo de servidor web, según interese.*

13.1.3 Interfaz de usuario

En la capa superior de la arquitectura encontramos las aplicaciones cliente que facilitarán al usuario el acceso a los servicios ofrecidos por el servidor de apli-

caciones. Estos clientes suelen ser programas muy ligeros, compuestos de una interfaz de usuario y la lógica imprescindible para invocar a los servicios remotos a través de peticiones HTTP o directamente con una comunicación TCP/IP.

Un cliente no se comunicará directamente con el servidor de datos, sino que delegará esta tarea en el servidor de aplicaciones. De esta manera se simplifica la distribución y aplicación del programa que tendrá que ejecutar el usuario, al no precisar software cliente del RDBMS ni otro *middleware* que resulta imprescindible para los clientes en una arquitectura cliente/servidor. En caso de operar sobre conjuntos de datos en el cliente, los componentes FireDAC aportarán toda la funcionalidad necesaria.

Aunque como desarrolladores que utilizan Delphi generalmente usaremos esta herramienta para construir los clientes de nuestra aplicación distribuida, en realidad un servidor de aplicaciones aceptando conexiones de tipo REST por HTTP es accesible de forma universal. Es factible, por lo tanto, crear una solución web basada en estándares HTML5/CSS3/Javascript como interfaz de usuario para consumir los servicios.

13.2 Comunicación entre interfaz y servidor de aplicaciones

Un servidor de aplicaciones DataSnap puede aceptar solicitudes mediante distintos canales de comunicación. Uno de ellos se mantiene únicamente por compatibilidad con versiones previas: DCOM. Los canales preferentes para nuevos proyectos son TCP/IP o HTTP. Será el servidor el que, dependiendo de los componentes que se incluyan y cómo se configuren, decida qué canal se usará, debiendo los clientes ajustarse a dicha configuración para poder consumir los servicios ofrecidos.

La comunicación a través de TCP/IP emplea un protocolo de más bajo nivel respecto a HTTP. Esto, no obstante, es algo que pasará prácticamente desapercibido para nosotros como desarrolladores. Al usar TCP/IP podremos seleccionar el puerto en el que el servidor de aplicaciones quedará a la escucha. Si dicho puerto no es estándar es probable que la comunicación se vea bloqueada por los cortafuegos instalados en cualquiera de los extremos o algún punto intermedio.

Existe, no obstante, la posibilidad de crear un *tunel* para canalizar la comunicación evitando estos obstáculos. HTTP, por el contrario, emplea un puerto que prácticamente siempre está abierto en todos los elementos de comunicación, ya que es el puerto estándar usado por la web.

DataSnap contempla el uso de filtros de comunicación. Estos pueden utilizarse para agregar funcionalidad como la compresión de datos a transferir o el cifrado de los datos. Los filtros pueden encadenarse de forma que la salida generada por uno es la entrada del siguiente, aplicando secuencias de transformaciones a los datos. Obviamente el cliente ha de utilizar la misma cadena de filtros pero en sentido inverso, obteniendo así la información que a la postre le resultará útil. A los filtros de compresión y cifrado, disponibles para su uso inmediato, es posible agregar otros propios, creados derivando una nueva clase de `TTransportFilter` e implementando los métodos correspondientes.

Otra fase importante en la comunicación entre la interfaz y el servidor de aplicaciones, concretamente al establecerla inicialmente, es la autenticación de usuarios y autorización de acceso a los servicios. DataSnap también cuenta con un componente específico para gestionar estas tareas. Básicamente hay que agregar un `TDSAuthenticationManager` al contenedor y gestionar adecuadamente los eventos `OnUserAuthenticate` y `OnUserAuthorize`.

13.3 Introducción a REST

En los últimos años la web, que en un tiempo se diseño únicamente como un mecanismo de hospedaje de contenidos hipervinculados, se ha convertido en una plataforma, una suerte de sistema operativo genérico, para ofrecer servicios de todo tipo. Aunque en principio se definieron formatos y protocolos específicos para tal fin, como SOAP (*Simple Object Access Protocol*) y XML/RPC (*Remote Procedure Call*), en la actualidad el mecanismo generalmente aceptado para esta tarea es REST (*Representational State Transfer*).

DataSnap cuenta con los componentes necesarios para implementar servidores de aplicaciones que atienden peticiones de tipo REST. Asimismo, Delphi también incorpora los elementos apropiados para consumir este tipo de servicios, ya estén implementados en Delphi o en cualquier otro lenguaje. En realidad REST permite acceder a un servicio con total transparencia respecto al sistema

operativo en que se este se ejecuta, la herramienta en que se ha desarrollado y otros parámetros dependientes de la plataforma.

REST tiene como pilar fundamental el protocolo HTTP, el mismo que se emplea habitualmente para acceder a cualquier página web. Siguiendo unas normas simples en cuanto a la nomenclatura de las rutas de acceso o URL, REST se basa en el uso de las distintas operaciones (*verbos*) contempladas por HTTP y los códigos de estado definidos en este estándar[1].

13.3.1 Composición de las rutas de acceso

Los URL de acceso a recursos mediante el protocolo HTTP se ajustan a un formato básico compuesto de los elementos indicados a continuación:

```
http://host:puerto/ruta?consulta
```

El término `host` será el nombre de un servidor o su dirección IP. Si se facilita un nombre se recurrirá al servidor DNS apropiado para resolverlo y obtener la dirección IP. El `puerto` no suele especificarse ya que por defecto se utiliza el puerto estándar `80`. Si se utilizase otro puerto habría que asegurarse de que es accesible. La `ruta` es la que identifica el recurso al que se quiere acceder, por ejemplo el nombre de la página. Esta parte es opcional, ya que el servidor web puede configurarse para facilitar un documento por defecto. Finalmente, el apartado `consulta` que sigue al carácter `?` puede contener múltiples pares del tipo `campo=valor` separados por el símbolo `&`.

Al implementar un servicio de tipo REST la ruta identificará el objeto sobre el que se quiere actuar. Este puede tener una relación jerárquica respecto a otros objetos que se vea reflejada en la ruta. Por ejemplo:

```
http://capri.es/pedido/linea
```

En este caso el URL haría referencia a una entidad, una línea de pedido, que forma parte de otra entidad, el pedido en sí mismo. La ruta puede incluir información adicional, según las necesidades del servicio. Para recuperar los datos de un pedido, por ejemplo, se podría entregar en la solicitud el identificador único de éste.

[1] Podemos encontrar la definición del estándar HTTP en `http://tools.ietf.org/html/rfc2616`.

Asimismo es posible que la entidad sea una colección de objetos, lo cual permite, por ejemplo, recuperar la lista de pedidos de un cliente con un URL como el siguiente:

```
http://capri.es/cliente/7362/pedidos
```

13.3.2 Verbos HTTP y operaciones

A diferencia de otras técnicas, como el antes mencionado XML/RPC, con REST la ruta no incluye el verbo que indica la operación a efectuar, sino el nombre del objeto sobre el que se actuará. La operación vendrá determinada por el tipo de solicitud HTTP que se emplee.

El estándar HTTP contempla ocho verbos distintos, de los cuales la mayoría de los servidores web suelen implementar solo dos o tres. Al desarrollar un servidor de tipo REST los verbos a implementar y las operaciones asociadas sobre la entidad designada en la ruta serían los siguientes:

- `POST`: Es el tipo de solicitud utilizado al enviar formularios HTML con información recogida del usuario. En un servidor REST representa la operación de creación de la entidad. Los datos de esta se facilitarán en el cuerpo de la solicitud HTTP.
- `GET`: La solicitud de tipo `GET` es la más común, sirviendo para recuperar cualquier página. En un servidor REST se usa para obtener el objeto indicado en la ruta. La información del objeto se obtendría como documento de respuesta.
- `PUT`: Mediante una solicitud de este tipo es posible modificar datos en el servidor, sustituyendo completamente la información asociada a una entidad.
- `DELETE`: Es la operación HTTP empleada para eliminar un recurso. Servirá para borrar la entidad designada en la ruta.

Como puede apreciarse, las operaciones contempladas son las habitualmente conocidas como CRUD (*Create, Read, Update, Delete*) en cualquier API que actúa sobre datos. Serían equivalentes a las sentencias de SQL `INSERT`, `SELECT`, `UPDATE` y `DELETE`.

NOTA

Las operaciones de un servidor REST no mantienen información de estado, lo que implica que son totalmente independientes entre sí. Todos los datos para completar la operación, por tanto, han de facilitarse en la propia solicitud. De esta forma el servidor no tiene que mantener información de estado de cada usuario que está accediendo a los servicios, lo cual contribuye a mejorar su rendimiento y escalabilidad.

13.4 Introducción a JSON

Si bien el formato con el que se transmitirán los datos entre cliente y servidor de aplicaciones, y viceversa, puede ser virtualmente cualquiera acordado entre ambas partes, cada vez es más frecuente utilizar JSON (*JavaScript Object Notation*), especialmente en combinación con servicios de tipo REST. JSON[2] es un estándar ECMA, al igual que JavaScript, pensado para facilitar el intercambio de datos entre aplicaciones, sin que importe la plataforma en que se ejecutan o el lenguaje en que estén desarrolladas.

En esta sección se describe la sintaxis básica de un documento JSON, primero, y se explica cómo trabajar con este tipo de información desde aplicaciones desarrolladas con Delphi, a continuación.

13.4.1 Sintaxis de JSON

JSON es un formato textual, no binario, con una estructura considerablemente simple, de ahí su éxito. A diferencia de otros formatos de intercambio de datos, como el conocido XML, JSON no precisa marcas de apertura y cierre, lo cual hace más fácil su interpretación y también reduce el tamaño de los documentos.

[2]Puedes encontrar la documentación completa sobre JSON, incluyendo una lista de bibliotecas que permiten trabajar con JSON a múltiples lenguajes de programación, en su sitio oficial: `http://json.org/`.

Un documento JSON tendrá la estructura básica mostrada a continuación:

```
objeto {
    "clave1": valor1,
    ...
    "claveN": valorN
}
```

Listado 13.1 Estructura básica de un documento JSON

Cada pareja `clave-valor` se separa con dos puntos. La `clave` es el nombre de cada campo que forma parte del objeto, debiendo facilitarse entrecomillado. El valor actual asociado a las claves puede ser de distintos tipos. Los considerados en la versión actual de JSON son:

- **Numérico**: Los valores numéricos se facilitan de forma directa, introduciendo la secuencia de dígitos y, opcionalmente, un signo como prefijo (+ o − para designar números positivos y negativos, respectivamente) y un punto decimal para separar la parte entera de la fraccionaria.
- **Cadena**: Las cadenas de caracteres han de introducirse entre comillas. El conjunto de caracteres válidos dependerá de la codificación e mpleada. Lo habitual es usar UTF-8.
- **Booleano**: Este tipo de campos solamente podrán tomar los valores constantes `true` y `false`.
- **Objeto**: El valor asociado a una clave de tipo complejo se facilita entre llaves, con una definición recursiva de la lista de campos y valores que lo componen.
- **Lista**: Una lista de valores se facilita entre corchetes, pudiendo ser sus elementos de cualquiera de los tipos previos.

Además de los ya mencionados, en un documento JSON también puede aparecer el valor `null` indicando que un cierto campo está vacío (no se le ha llegado a asignar un valor en ningún momento).

El siguiente ejemplo muestra un documento JSON con los datos de un hipotético cliente, incluyendo campos de distintos tipos. El campo `Pedidos` es una

lista de objetos, de ahí que se utilicen corchetes para contener la lista de elementos y llaves para delimitar los campos de cada elemento.

```
{
  "Cliente": {
    "Nombre": "Programación de
    Aplicaciones Delphi con Acceso a Bases de
    Datos", "ClienteHabitual":
    true,
      "TipoVia": "Calle",
      "Via": "Bailén",
      "Numero": 56
    },
    "Pedidos": [
      {
        "Fecha": "12/6/2014",
        "Importe": 324.65
      },
      {
        "Fecha": "8/10/2014",
        "Importe": 187.43
      }
    ]
  }
}
```

Listado 13.2 Documento JSON de ejemplo con datos de un cliente

NOTA

En este ejemplo se han incluido saltos de línea y sangrados para hacer más legible la estructura del documento. En la práctica todo ese espacio sobrante puede eliminarse para hacer más compacta la representación y reducir la cantidad de información a transferir.

13.4.2 Trabajar con JSON en Delphi

Ahora que conocemos los fundamentos sobre JSON, veamos cómo podemos trabajar con este tipo de información desde una aplicación Delphi. Para ello recurriremos a las clases definidas en el módulo `System.JSON`. Al ser parte de la RTL, estos servicios están disponibles para cualquier tipo de proyecto Delphi, incluyendo programas de consola, aplicaciones basadas en VCL y en FMX, servidores, etc.

En la base de estos servicios se encuentra la clase `TJSONValue`. Esta representa cualquier valor en un documento JSON, aportando los métodos genéricos `GetValue` y `TryGetValue`. De ella descienden las siguientes clases especializadas:

- `TJSONNumber`: Representa un número cualquiera. Las propiedades `AsInt`, `AsInt64` y `AsDouble` permiten recuperar el valor en formato entero, entero de 64 bits o punto flotante.
- `TJSONString`: Representa una cadena JSON. El contenido de dicha cadena se recupera mediante el método `Value`. La versión JSON de la cadena, entrecomillada, se obtiene con el método `ToString`.
- `TJSONTrue` y `TJSONFalse`: Estos dos tipos representan los valores `true` y `false` de JSON.
- `TJSONObject`: Es la clase asociada a un objeto JSON. Cuenta con propiedades indexadas, como `Values` y `Pairs`, que facilitan el acceso a los pares `clave-valor` existente en el objeto. Asimismo pueden agregarse y eliminarse pares mediante los métodos `AddPair` y `RemovePair`. También es capaz de decodificar una cadena JSON introduciendo toda la información en el objeto, mediante el método `Parse`. Análogamente, el método `ParseJSONValue` facilita la recuperación de valores a partir de secuencias de bytes.
- `TJSONArray`: Representa una lista de valores JSON. Con las propiedades `Count` e `Items` se obtiene el número de elementos existentes en la lista y se accede a los valores de cada uno. También cuenta con métodos como `AddElement`, `Remove` y `GetEnumerator` que facilitan la adición, eliminación y recorrido de los elementos.

- `TJSONNull`: Representa el valor `null` de JSON.

Todas estas clases cuentan con métodos que facilitan la *serialización* de la información que contienen, como `ToBytes` y `ToString`. El primero genera una representación en forma de secuencia de bytes, mientras que el segundo produce la cadena JSON correspondientes. Con el método `EstimatedByteSize` se puede obtener el tamaño mínimo que habría que reservar para alojar la información devuelta por el método `ToBytes`.

Los pares `clave-valor` se representan mediante objetos `TJSONPair`, cuyo constructor acepta dos parámetros con los que se establece el nombre del campo y su valor. Estos pares pueden después ser agregados a objetos de tipo `TJSONObject` usando el citado método `AddPair`.

Supongamos que queremos crear desde Delphi la estructura de datos necesaria para enviar información de un cliente en formato JSON, generando un documento equivalente al del Listado 13.2. Lo primero que haríamos sería crear los objetos de primer nivel, encargados de contener los datos propiamente dichos, usando para ello el código mostrado a continuación:

```
1  ...
2  uses
3    System.SysUtils, System.JSON;
4
5  var
6    cliente, direccion, pedido1, pedido2: TJSONObject;
7    pedidos: TJSONArray;
8
9  begin
10   cliente := TJSONObject.Create;
11   direccion := TJSONObject.Create;
12   pedido1 := TJSONObject.Create;
13   pedido2 := TJSONObject.Create;
14   pedidos := TJSONArray.Create;
15
16  ...
```

Listado 13.3 Creación de los objetos de primer nivel

EJEMPLO 13.1 `JSONDoc.dproj`

Puedes encontrar este proyecto completo en la carpeta `JSONDoc`. No se necesita ajuste alguno para poder ejecutarlo. Es una aplicación de consola.

A continuación introduciríamos en cada uno de esos objetos la información apropiada, usando para ello el método `AddPair` en el caso de los `TJSONObject`. Por ejemplo, para componer la dirección usaríamos el siguiente código:

```
...
with direccion do begin
  AddPair(TJSONPair.Create(
    TJSONString.Create('TipoVia'),
    TJSONSTring.Create('Calle')
  ));
  AddPair(TJSONPair.Create(
    TJSONString.Create('Via'),
    TJSONSTring.Create('Bailén')
  ));
  AddPair(TJSONPair.Create(
    TJSONString.Create('Numero'),
    TJSONNumber.Create(56)
  ));
end;
...
```

Listado 13.4 Introducción de los datos de dirección

Los datos de los pedidos se agregarían a los respectivos objetos `pedido1` y `pedido2`, de tipo `TJSONObject`, siendo agregados a continuación a la lista, representada por el objeto `TJSONArray`, mediante el método `AddElement`, tal y como se muestra a continuación:

```
...
with pedido1 do begin
  AddPair(TJSONPair.Create(
    TJSONString.Create('Fecha'),
    TJSONSTring.Create('12/6/2014')
  ));
  AddPair(TJSONPair.Create(
    TJSONString.Create('Importe'),
    TJSONNumber.Create(324.65)
  ));
end;

with pedido2 do begin
  AddPair(TJSONPair.Create(
    TJSONString.Create('Fecha'),
    TJSONSTring.Create('8/10/2014')
  ));
  AddPair(TJSONPair.Create(
    TJSONString.Create('Importe'),
    TJSONNumber.Create(187.43)
  ));
end;

pedidos.AddElement(pedido1);
pedidos.AddElement(pedido2);
...
```

Listado 13.5 Composición de la lista de pedidos

El último paso es la creación del objeto raíz, encargado de contener algunos datos simples y los demás objetos. El código necesario para ello es el siguiente:

```
...
with cliente do begin
  AddPair(TJSONPair.Create(
    TJSONString.Create('Nombre'),
    TJSONSTring.Create('Francisco␣Charte')
```

```
    ));
    AddPair(TJSONPair.Create(
      TJSONString.Create('Saldo'),
      TJSONNumber.Create(512.08)
    ));
    AddPair(TJSONPair.Create(
      TJSONString.Create('ClienteHabitual'),
      TJSONTrue.Create
    ));
    AddPair(TJSONPair.Create(
      TJSONString.Create('Direccion'),
      direccion
    ));
    AddPair(TJSONPair.Create(
      TJSONString.Create('Pedidos'),
      pedidos
    ));
  end;
  ...
```

Listado 13.6 Creación del documento raíz

Para obtener el documento JSON correspondiente al objeto `cliente`, que contiene toda la información previa, no tenemos más que invocar al método `ToString`. El código de ejemplo siguiente generaría la salida mostrada en la Figura 13.2. Como puede apreciarse en la cadena no se introducen saltos de línea ni espaciado adicional, obteniendo así una representación compacta.

```
  ...
  WriteLn('Representación␣JSON␣del␣TJSONObject' +
    #13#10#10 + cliente.ToString +
    #13#10#10 + 'Pulsa␣<Intro>');

  ReadLn;
  ...
```

Listado 13.7 Obtención del JSON correspondiente al objeto

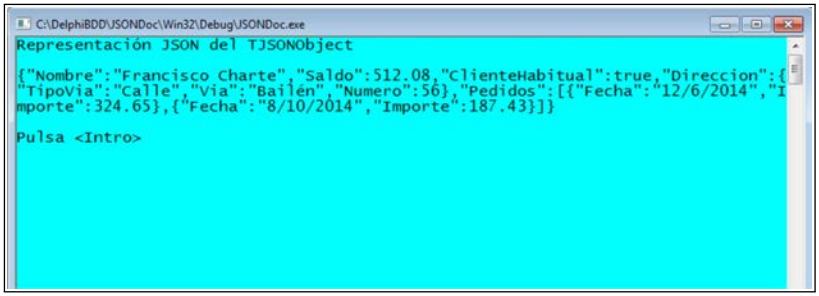


Figura 13.2 DOCUMENTO JSON OBTENIDO A PARTIR DEL `TJSONOBJECT`.

En el anterior ejemplo hemos aportado la información desde el código Delphi y generado el documento JSON, presumiblemente para enviarlo a un servidor. También será preciso poder llevar a cabo la operación inversa, siendo esta la finalidad del método `Parse`. Este acepta como argumento la secuencia de bytes del documento JSON obtenido del servidor, generando internamente las estructuras de datos adecuadas en el objeto `TJSONObject`.

Una vez que se cuenta con el citado objeto, usaríamos métodos como `Values`, `GetValue` y `Value` para acceder a la información que pudiera interesarnos. En el siguiente ejemplo se muestra cómo generar el `TJSONObject` a partir de una cadena literal (en la práctica sería una variable recibida como respuesta del servidor) y cómo recuperar un dato concreto: el nombre de la vía correspondiente a la dirección del cliente:

```
1  cliente := TJSONObject.Create;
2
3  cliente.Parse(BytesOf('{␣"Cliente":␣' +
4     '{␣"Nombre":␣"Francisco␣Charte",␣' +
5     '␣"ClienteHabitual":␣true,␣' +
6     '␣"Saldo":␣512.08,␣' +
7     '␣"Direccion":␣' +
8     '{␣"TipoVia":␣"Calle",␣' +
9     '␣"Via":␣"Bailén",␣' +
10    '␣"Numero":␣56␣},␣' +
11    '␣"Pedidos":␣[␣{␣' +
```

```
12      '␣"Fecha":␣"12/6/2014",␣' +
13      '␣"Importe":␣324.65␣},{␣' +
14      '␣"Fecha":␣"8/10/2014",␣' +
15      '␣"Importe":␣187.43␣}]}}'
16    ), 0);
17
18    WriteLn('Vía␣de␣la␣dirección:␣' +
19      cliente.Values['Cliente'].GetValue<TJSONObject>('↙
          ↳ Direccion').Values['Via'].Value +
20      #13#10#10 + 'Pulsa␣<Intro>'
21    );
22
23    ReadLn;
```

Listado 13.8 Generación del objeto a partir de un documento JSON

13.5 Resumen

Este capítulo nos ha servido para obtener una visión general sobre la arquitectura y comunicación entre las distintas partes que componen una solución distribuida. También nos ha permitido conocer aspectos básicos sobre tecnologías que se utilizan en este campo, como son REST, para la gestión de peticiones de servicios por parte de los clientes, y JSON, como técnica de intercambio de datos multi-plataforma.

Los capítulos siguientes se apoyarán en lo que hemos aprendido en este a fin de abordar el desarrollo de servidores DataSnap y clientes capaces de consumir los servicios ofrecidos por estos.

Capítulo 14

SERVICIOS DATASNAP

Con DataSnap es posible crear tanto servidores de aplicaciones *tradicionales*, de forma que los clientes usan invocaciones de tipo RPC para desencadenar la ejecución de una acción, como servicios de tipo REST. En este capítulo nos ocuparemos del desarrollo y consumo de los primeros, dejando para el capítulo siguiente los segundos.

Como tendremos oportunidad de comprobar en los ejercicios propuestos, una buena parte del trabajo asociado a la creación de un servidor de aplicaciones DataSnap y los clientes que consumirán sus servicios está a cargo de asistentes. Estos generarán el esqueleto, con los componentes y código básico, sobre el que

implementaremos la funcionalidad que queremos tenga el servidor. De manera análoga, para la aplicación cliente el asistente aportará los elementos necesarios para facilitar la comunicación con los métodos expuestos por el servidor.

En las secciones siguientes se explica detalladamente el proceso para crear un servidor DataSnap, comenzando por un caso básico para avanzar a continuación hacia servicios que devuelven conjuntos de datos. Asimismo se describirá cómo consumir esos servicios desde los clientes.

14.1 Aspectos fundamentales de DataSnap

Comenzamos introduciendo algunos de los aspectos básicos relativos al desarrollo de servicios con DataSnap. Estos son comunes a todo servidor creado con DataSnap, por lo que son conceptos que nos resultarán útiles a lo largo de todo el capítulo y también en el capítulo siguiente.

14.1.1 Aplicación contenedora

Un servidor DataSnap puede estar alojado en distintos tipos de aplicación. La aplicación actuará, en cierta manera, como contenedora del servidor. Los tipos de contenedores posibles son:

- **Aplicaciones ejecutables de tipo consola**: El servidor contará con una interfaz basada en texto.
- **Aplicaciones ejecutables con interfaz gráfica de usuario**: Se puede utilizar tanto VCL como FMX para componer dicha interfaz de usuario.
- **Servicios Windows**: El servidor se instala como un servicio y es el sistema, a través de herramientas como la consola **SERVICIOS**, el que lo controla. El servidor no contará con interfaz alguna, usándose habitualmente el registro de eventos del sistema para comunicarse con el usuario. Existe la posibilidad de crear una herramienta complementaria para supervisar y controlar el servicio.

- **Módulos de servidor web**: El servidor DataSnap puede estar alojado en un módulo ISAPI (*Internet Information Server*) o un módulo de Apache, operando solidariamente con el respectivo servidor web.

En función de cómo deseemos que se ejecute el servidor de aplicaciones, y cuál sea la interfaz de comunicación con el encargado de administrarlo, escogeríamos un tipo de contenedor u otro.

14.1.2 Elementos del servidor DataSnap

El objetivo principal de un servidor DataSnap es ofrecer un conjunto de servicios a los clientes. Estos serán implementados por una o varias clases que pueden derivar de `TComponent`, `TDataModule` o `TDSServerModule`, según interese. Para que los métodos aportados por dichas clases puedan ser invocados desde los clientes es necesario que el servidor DataSnap cuente con los siguientes elementos:

- **Información de tipo de las clases de servicio**: El servidor DataSnap precisa de información RTTI (*Run Time Type Information*) extendida sobre cada una de las clases que aportan servicios, concretamente sobre los parámetros de los métodos y la forma de gestionarlos. Para generar dicha información, es necesario activar la directiva `$METHODINFO`[1] antes de definir la clase, desactivándola opcionalmente al final.
- **Control del ciclo de vida de las clases de servicio**: Cuando un cliente solicite un cierto servicio será preciso disponer de una instancia de la clase que lo implementa. La responsabilidad de gestionar el ciclo de vida de estas clases recae en el componente `TDSServerClass`.
- **Control del estado del servidor**: El contenedor del servidor DataSnap puede ser ejecutado manualmente, ponerse en marcha como un servicio o ser iniciado por el servidor web. En cualquier caso, que el contenedor del servidor esté ejecutándose no implica automáticamente que el propio servidor esté activo. El estado de éste lo controlará el componente `TDSServer`.

[1] La activación de esta directiva incrementará el tamaño del ejecutable, de ahí que normalmente se use exclusivamente al definir las clases de servicio y se mantenga desactivada para el resto del código.

- **Comunicación con los clientes**: Dependiendo del tipo de contenedor en que se encuentre el servidor, este deberá aportar los componentes necesarios para que el cliente puede hacer llegar sus solicitudes. Para poder quedar a la escucha en un puerto TCP/IP y responder solicitudes que lleguen por dicho canal, por ejemplo, incluiríamos en el servidor un componente `TDSTCPServerTransport` adecuadamente configurado. Otras clases de transporte son `TDSHTTPServerTransport`, para peticiones HTTP, y `TCustomDSRESTServerTransport`, para servidores de tipo REST. Todas estas clases derivan de `TDSServerTransport`, clase que actúa como interfaz común para la capa de transporte.

14.1.3 Componentes en el servidor

Al desarrollar un servidor de aplicaciones con DataSnap recurriremos a un conjunto de componentes específico, alojado en la página **DATASNAP SERVER** de la Paleta de herramientas de Dephi (véase la Figura 14.1). Los esenciales para cualquier proyecto son los siguientes:

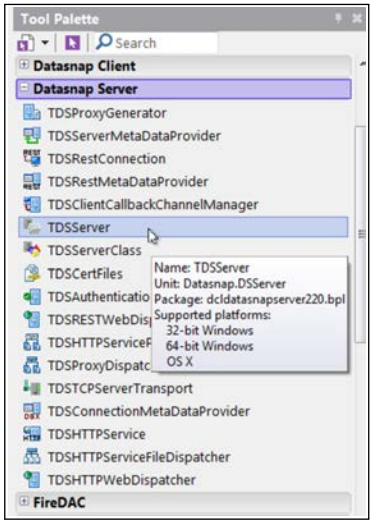


Figura 14.1 COMPONENTES PARA SERVIDORES DATASNAP

- `TDSServer`: Cada servidor DataSnap incorporará un componente de este tipo. Su finalidad es controlar el estado del servidor, encargándose de la creación y destrucción del resto de componentes, incluyendo los de comunicación y los `TDSServerClass` existentes. La propiedad `AutoStart` determina si el servidor se iniciará automáticamente al ejecutarse el contenedor. De no ser así, tendremos que usar los métodos `Start` y `Stop` para iniciarlo y detenerlo, respectivamente. Mediante las propiedades `ChannelResponseTimeout` y `ChannelQueueSize` se establece el tiempo de espera para que un cliente responda antes de darlo por desconectado y el tamaño de la cola de mensajes para cada cliente.

- `TDSServerClass`: Contaremos con al menos una copia de este componente en el servidor. Su finalidad es crear instancias de la clase que ofrece los servicios. El enlace entre el componente y la clase se establece en el método asociado al evento `OnGetClass` del primero, asignando la clase al parámetro `PersistentClass`. El ciclo de vida de la clase de servicios vendrá determinado por el valor asignado a la propiedad `LifeCycle` del `TDSServerClass`. Con 3 valores posibles:

  - `Invocation`: Se creará un objeto de la clase de servicio para cada llamada recibida de los clientes, dando lugar a una configuración de tipo *stateless*, en la que no se mantiene información de estado entre llamadas.
  - `Server`: Se creará un único objeto de la clase de servicio, al poner en marcha el servidor, de forma que éste será compartido por todos los clientes que efectúen llamadas.
  - `Session`: Se creará un objeto independiente para cada cliente, manteniéndolo activo mientras la conexión esté viva. De esta forma se mantiene información de estado individual por cada sesión de trabajo.

- `TDSTCPServerTransport`: Para los servidores propuestos como ejemplo en este capítulo usaremos este componente de transporte, habilitando así la comunicación por `TCP/IP`. El puerto que se quedará a la escucha[2] lo establece la propiedad `Port`. Este componente está preparado para gestionar múltiples solicitudes simultáneamente mediante el uso de *threads*.

[2]El componente se activará y desactivará mediante los métodos `Start` y `Stop`. Estos serán invocados automáticamente por el componente `TDSServer` cada vez que el servidor inicie y detenga, respectivamente.

Las propiedades `MaxThreads` y `PoolSize` son las encargadas de configurar el número máximo de hilos de ejecución y el tamaño del *pool* o depósito de hilos reutilizables.

Además de estos componentes, que son los fundamentales, es posible incluir otros en el servidor dependiendo de la funcionalidad que quiera agregarse: capacidad de autenticación y autorización, cifrado y compresión del flujo de datos, conversión a JSON, conexión a RDBMS, etc. En los ejercicios desarrollados más adelante, en este y el capítulo siguiente, conoceremos algunos de dichos componentes.

14.2 Un servicio simple

Veamos en la práctica cómo desarrollar un servidor DataSnap sencillo. El objetivo de este servidor será facilitar números aleatorios en un rango indicado, asumiendo que contamos con un mecanismo de alta calidad de generación de números aleatorios que presumiblemente interesaría a nuestros clientes.

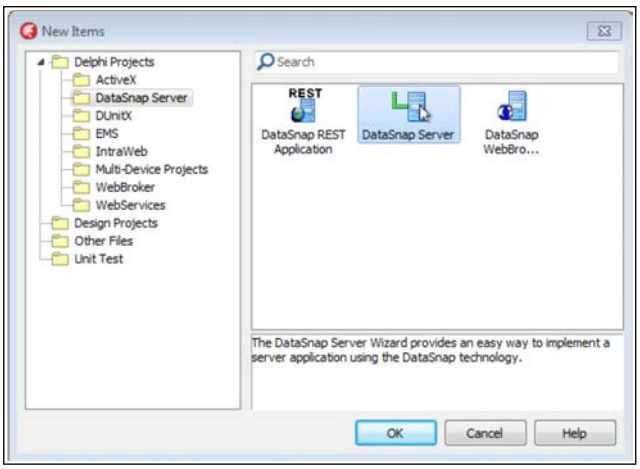


Figura 14.2 OPCIÓN PARA INICIAR EL ASISTENTE DE SERVIDOR DATASNAP.

La mayor parte del trabajo se completará mediante un asistente. Pondremos este en marcha abriendo el cuadro de diálogo **NEW ITEMS**, usando la opción **FILE—NEW—OTHER**, y seleccionando el elemento **DATASNAP SERVER** de la rama **DATASNAP SERVER** (véase la Figura 14.2). El número de pasos del asistente es variable, dependiendo de las opciones que vayan eligiéndose. A continuación se describen los que usaremos para este ejemplo concreto:

1. Lo primero será elegir el tipo de contenedor en el que va a residir el servidor DataSnap. Como se aprecia en la Figura 14.3, el asistente nos ofrece tres opciones: una aplicación basada en formularios, una aplicación de consola o un servicio. Elegiremos la primera y haremos clic en el botón **NEXT** para avanzar.

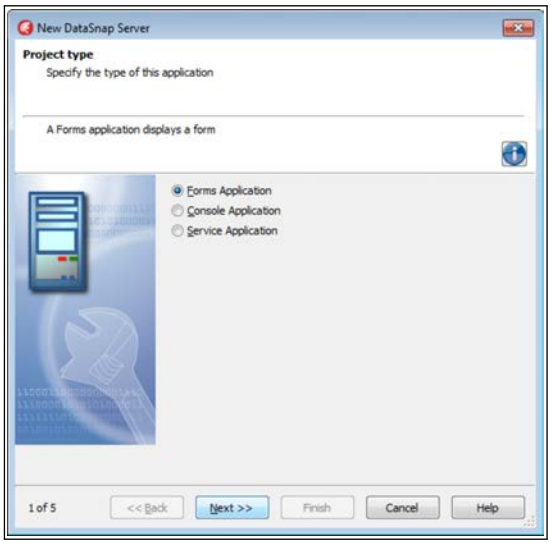


Figura 14.3 SELECCIÓN DEL TIPO DE CONTENEDOR

2. Al haber seleccionado **FORMS APPLICATION** en el paso previo, en este podremos elegir entre utilizar VCL o FMX como biblioteca de componentes para el formulario. Escogeremos la segunda opción.

3. En este paso podremos elegir la funcionalidad que deseamos incorporar ya de partida en el servidor. Como puede verse en la Figura 14.4 la lista de opciones está agrupada en varias categorías: protocolos de comunicación, servicios de autenticación, filtros, etc. En principio podemos dejar la selección que aparece por defecto, al tratarse de un ejercicio simple.

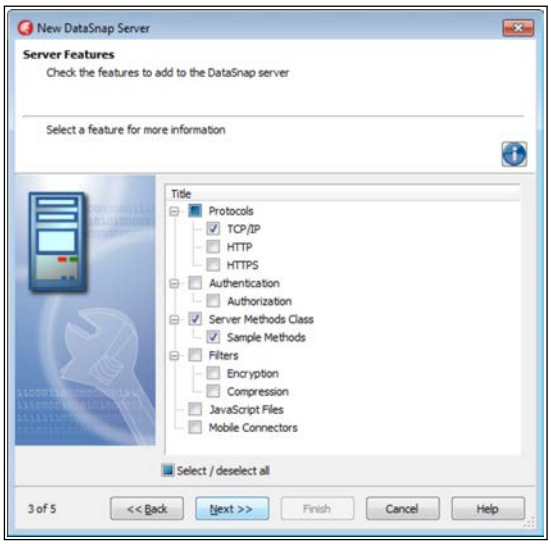


Figura 14.4 COMPONENTES A INCLUIR EN EL SERVIDOR

4. Al haber seleccionado TCP/IP como protocolo de transporte, será preciso elegir el puerto en el que el servidor quedará a la escucha. El seleccionado por defecto es el 211, pero podemos cambiarlo según nuestras necesidades. Este paso del asistente también nos permite (véase la Figura 14.5) comprobar si el puerto está disponible, así como buscar un puerto disponible en el sistema.

5. En este último paso del asistente tendremos que indicar cuál será la clase de la que se derive nuestra clase de servicio, aquella en la que implementaremos los métodos accesibles para los clientes. Las opciones posibles son tres: `TComponent`, `TDataModule` y `TDServerModule`. En este caso concreto elegiremos la primera. Dado que no vamos a trabajar con conjuntos

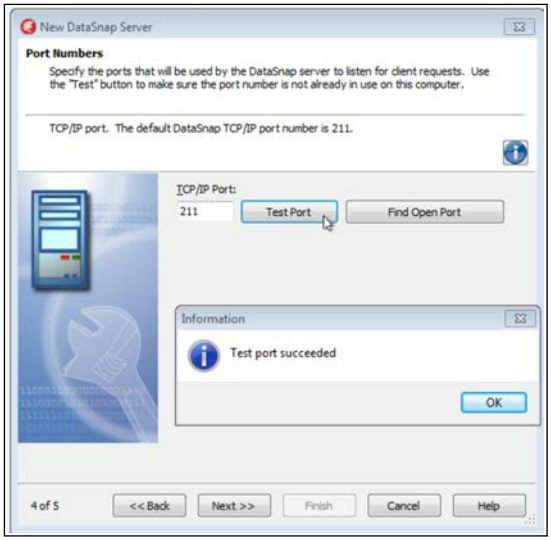


Figura 14.5 CONFIGURAMOS Y PROBAMOS EL PUERTO DE COMUNICACIÓN

de datos, no precisamos la funcionalidad aportada por las otras. Hacemos clic en **FINISH** para proceder a generar el proyecto.

EJEMPLO 14.1 `RandomService.dproj`

Puedes encontrar este proyecto completo en la carpeta `RandomService`. Al ejecutarlo puede ser preciso abrir el puerto TCP/IP en el que el servidor quedará a la escucha.

El proyecto generado por el asistente constará de tres módulos: el formulario FMX que actúa como interfaz de usuario, un módulo con la clase que implementa los métodos ofrecidos como servicios y un tercer módulo correspondiente al contenedor. Este será una clase derivada de `TDataModule` con los elementos que pueden verse en la Figura 14.6.

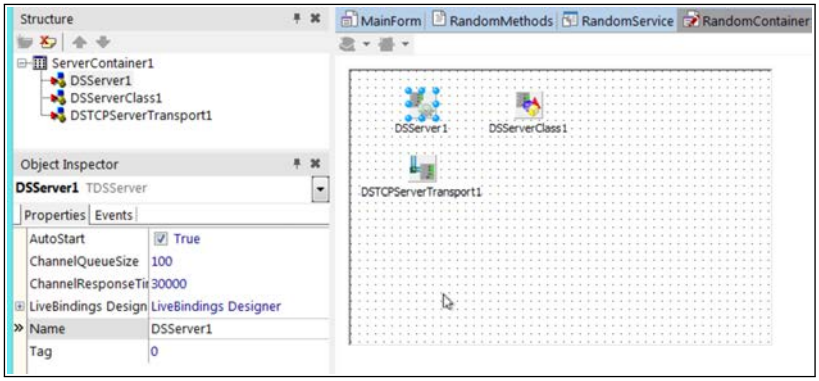


Figura 14.6 COMPONENTES EN EL MÓDULO CONTENEDOR

En el módulo contenedor encontraremos el código que se encarga de enlazar el componente `TDSServerClass` con la clase que expone métodos como servicios. Esta acción se efectúa en respuesta al evento `OnGetClass` de dicho componente, tal y como se explicó antes:

```
1 procedure TServerContainer1.DSServerClass1GetClass(
2   DSServerClass: TDSServerClass; var PersistentClass: ↙
      ↳ TPersistentClass);
3 begin
4   PersistentClass := RandomMethods.TRandomMethods;
5 end;
```

Listado 14.1 Vinculación entre el `TDSServerClass` y la clase de servicio

14.2.1 Implementación del servicio

En este momento tenemos un proyecto que es un esqueleto de servidor DataSnap, incluyendo dos servicios básicos en caso de que no hayamos desmarcado la opción **SAMPLE METHODS** en el tercer paso del asistente. Este esqueleto consta de un módulo conteniendo la clase, derivada de `TComponent`, con los

métodos que serán accesibles para los clientes. Vamos a completar esta clase implementando el servicio ofrecido por nuestro servidor DataSnap. Para ello reproduciremos los pasos indicados a continuación:

- Lo primero que haremos será cambiar el nombre de la clase que aporta los métodos expuesto por el servidor, a la que llamaremos `TRandomMethods`. Completaremos su definición agregando a la sección pública un método adicional: `NextRand`. Este tomará dos parámetros enteros como entrada y devolverá un valor del mismo tipo. La definición de la clase, incluyendo los dos métodos de ejemplo incluidos por el asistente, quedaría así:

```
type
{$METHODINFO ON}
  TRandomMethods = class(TComponent)
  private
    { Private declarations }
  public
    { Public declarations }
    function EchoString(Value: string): string;
    function ReverseString(Value: string): string;
    function NextRand(FromN: integer; ToN: integer): ↙
        ↳ integer;
  end;
{$METHODINFO OFF}
```

Listado 14.2 Agregamos la definición del método `NextRand`

- A continuación agregaremos a la sección de implementación del módulo el código del nuevo método. Como se aprecia a continuación, nos limitamos a usar la función `Random` para devolver un nuevo valor aleatorio:

```
function TRandomMethods.NextRand;
begin
  Result :=  Random(ToN-FromN) + FromN;
end;
```

Listado 14.3 Implementación del método `NextRand`

Completado este procedimiento tenemos finalizado el desarrollo de nuestro primer servidor DataSnap. Podemos ejecutarlo directamente desde el entorno de Delphi, preferentemente con la opción de ejecución sin depuración. Esto nos permitirá minimizar la ventana y continuar trabajando en el IDE, a fin de probar el funcionamiento del servidor. Al lanzar este por primera vez probablemente aparezca una notificación del cortafuegos de Windows (véase la Figura 14.7), preguntándonos si deseamos permitir la comunicación demandada por este programa.

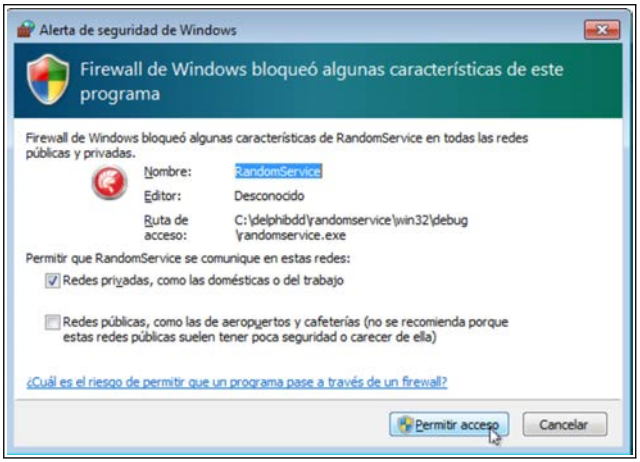


Figura 14.7 EL CORTAFUEGOS DE WINDOWS NOTIFICA QUE EL SERVIDOR DEMANDA COMUNICACIÓN POR RED

14.2.2 Acceso al servicio desde el IDE

A pesar de que el servidor DataSnap que hemos desarrollado no expone dato alguno, solo una función a la que es posible invocar de forma remota, podemos utilizar el Explorador de datos del IDE de Delphi para comprobar que, en efecto, el servidor está en funcionamiento.

Lo primero que debemos hacer es crear una nueva conexión en el Explorador de datos, concretamente en la rama **FIREDAC—DATASNAP SERVER**. En el

cuadro de diálogo de parámetros nos aseguramos de elegir el controlador `DS` y establecer el puerto de comunicación en el que el servidor está a la escucha, como se muestra en la Figura 14.8. Podemos hacer clic en el botón **TEST** para comprobar que existe conexión con el servidor (siempre que este se encuentre en ejecución).

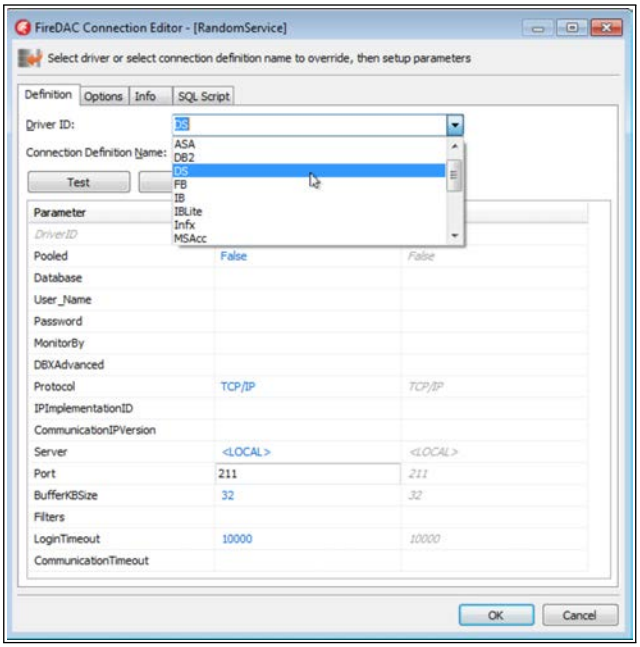


Figura 14.8 DEFINIMOS LA CONEXIÓN PARA ACCEDER AL SERVIDOR DATASNAP

Asumiendo que tenemos definida la conexión y el servidor está ejecutándose, podemos abrir el nodo correspondiente en el Explorador de datos y, a continuación, desplegar la rama **PROCEDURES**. Ahí encontraremos los métodos expuestos como servicios por el servidor, así como su lista de parámetros (véase la Figura 14.9). De esta manera tan inmediata y sencilla nos aseguramos de que el servidor DataSnap recién desarrollado opera como debería, poniendo a disposición de los clientes el método `NextRand` que implementábamos anteriormente.

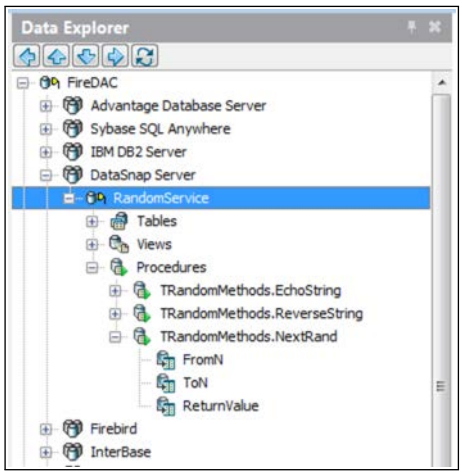


Figura 14.9 EXAMINAMOS EL SERVICIO DESDE EL EXPLORADOR DE DATOS

NOTA

En caso de que el servidor DataSnap contase con mecanismo de autenticación de usuarios, al definir la conexión anterior habría que facilitar las credenciales necesarias para poder conectar.

14.2.3 Un consumidor simple

A la hora de desarrollar un cliente DataSnap, capaz de consumir los servicios ofrecidos por el servidor, contamos con distintas alternativas. Una de ellas consiste en utilizar FireDAC, concretamente el componente `TFDStoredProc`, para llamar a las funciones como si de procedimientos almacenados se tratase. Otra posibilidad es generar un módulo cliente DataSnap, usando para ello el asistente correspondiente, que haga aparecer como locales los métodos remotos, facilitando así su invocación. Veamos cómo construiríamos cada tipo de cliente.

Consumidor usando FireDAC

En el apartado previo hemos comprobado cómo es posible acceder al servidor DataSnap desde el Explorador de servidores, previa definición de la conexión correspondiente. El servidor puede ser tanto local como remoto, solo hemos de cambiar el contenido de una propiedad para establecer su dirección IP.

Teniendo el método en el Explorador de servidores, todo lo que tenemos que hacer es arrastrarlo hasta el formulario de nuestra aplicación para agregar un `TFDConnection` y un `TFDStoredProc` adecuadamente configurados. Tendremos que añadir también un `TFDPhysDSDriverLink` y el habitual componente `TFDGUIxWaitCursor`.

Agregaremos al formulario dos controles `TEdit`, a fin de poder introducir los parámetros de invocación del método `NextRand`; un `TButton` que desencadenará la invocación y varios `TLabel`, todo ello distribuido en un `TGroupBox` tal y como se muestra en la Figura 14.10.

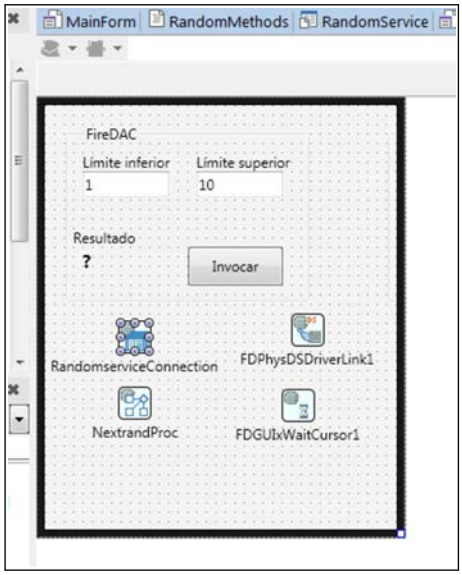


Figura 14.10 COMPOSICIÓN DE LA INTERFAZ Y COMPONENTES DEL CONSUMIDOR

Hacemos doble clic sobre el `TButton`, abriendo el método correspondiente a su evento `OnClick`. Este será el punto en el que facilitemos al componente `TFDStoredProc` los parámetros de entrada, ejecutemos el método remoto y obtengamos el resultado. Dado que el servicio no devuelve un conjunto de datos, sino un valor discreto, usaremos el método `ExecProc` en lugar del método `Open` del `TFDStoredProc`. El código del evento será el siguiente:

```
1 procedure TfrmMain.Button1Click(Sender: TObject);
2 begin
3   with NextrandProc do begin
4     ParamByName('FromN').Value := StrToInt(edInferior.Text);
5     ParamByName('ToN').Value := StrToInt(edSuperior.Text);
6     ExecProc;
7     lblResultado.Text := ParamByName('ReturnValue').AsString↙
          ↳ ;
8   end;
9 end;
```

Listado 14.4 Invocación del método remoto como procedimiento almacenado

No necesitamos más para ejecutar la aplicación cliente y, asumiendo que el servidor esté en funcionamiento, comprobar cómo al pulsar el botón van obteniéndose números aleatorios en el rango solicitado. En este caso la operación ofrecida como servicio es muy simple, en un proyecto real podría realizarse virtualmente cualquier trabajo y generar un resultado más complejo.

EJEMPLO 14.2 `RandomClient.dproj`

Puedes encontrar este proyecto completo en la carpeta `RandomClient`. Para poder ejecutarlo es necesario tener en funcionamiento el servidor desarrollado anteriormente, así como tener abiertos los puertos de comunicación correspondientes.

Consumidor usando módulo cliente DataSnap

La alternativa al procedimiento que acaba de explicarse, basado en FireDAC, consiste en utilizar un asistente de Delphi capaz de comunicarse con el servidor, obtener información sobre los métodos que este ofrece y generar el código necesario para facilitar el acceso a los mismos como si fuesen locales. Este asistente se pone en marcha mediante la opción **DATASNAP CLIENT MODULE** del grupo **DATASNAP SERVER** (véase la Figura 14.11).

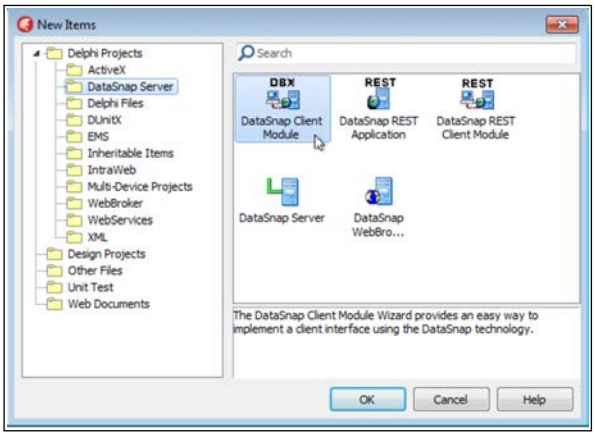


Figura 14.11 LANZAMOS EL ASISTENTE PARA GENERAR EL MÓDULO CLIENTE

Sobre el mismo proyecto anterior, con un cliente que consume el servicio mediante FireDAC, vamos a poner en marcha el mencionado asistente. Este consta de los pasos indicados a continuación:

1. Tendremos que indicar si el servidor DataSnap es local, se ejecuta en la misma máquina en la que estamos desarrollando, o remoto. Esto afectará a la información solicitada más adelante, en el cuarto paso.

2. A continuación tendremos que indicar al asistente el tipo de servidor DataSnap con el que queremos conectar. Como se aprecia en la Figura 14.12 existen múltiples opciones. Elegiremos la primera, que corresponde a un servidor estándar en un contenedor que puede ser aplicación de consola o basada en formularios.

Figura 14.12 ESPECIFICAMOS EL TIPO DE SERVIDOR DATASNAP AL QUE SE QUIERE ACCEDER

3. En el tercer paso tendremos que indicar si el servidor contempla la comunicación mediante el protocolo HTTP o bien usando TCP/IP. Esta segunda opción es la elegida por defecto y la que nos interesa en este ejercicio.

4. El último paso del asistente nos solicita los parámetros necesarios para poder establecer comunicación con el servidor DataSnap. Estos incluirán el nombre o dirección IP del servidor (en caso de que sea remoto), el puerto en que está a la escucha y, en caso necesario, las credenciales de autenticación en el mismo. Antes de finalizar el asistente podemos comprobar si los parámetros introducidos son apropiados, verificando la conexión con el servidor como se hace en la Figura 14.13.

Al finalizar, el asistente agregará al proyecto dos nuevos módulos de código. Uno de ellos, con el nombre `ClientModuleUnit1` por defecto, contiene una clase derivada de `TDataModule` con un componente `TSQLConnection`.

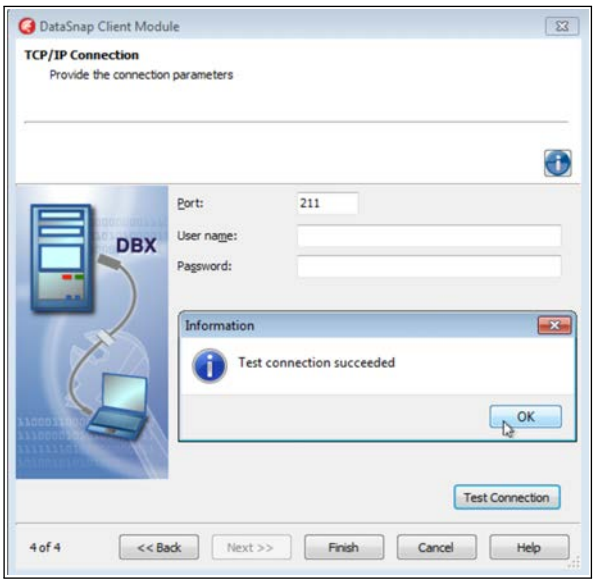


Figura 14.13 FACILITAMOS LOS PARÁMETROS DE COMUNICACIÓN CON EL SERVIDOR

Este se encuentra configurado para facilitar el acceso al servidor, aunque utilizando dbExpress en lugar de FireDAC. La clase cuenta, además, con una propiedad que devuelve una instancia de la clase definida en el segundo módulo, llamado `ClientClassesUnit1` por defecto. Dicha clase tendrá, por cada método ofrecido como servicio por el servidor, un método con idéntico nombre, lista de parámetros de entrada y de salida. Esto permite invocar a los métodos remotos con la misma comodidad que si fuesen locales. En realidad, internamente dichos métodos se ocupan de convertir los argumentos en parámetros de llamada al procedimiento remoto, como hacíamos en el ejemplo anterior de forma manual.

Asumiendo que hemos guardado los módulos, habiendo renombrado las clases como `TRandomModule` y `TRandomMethodsClient`, volveremos al formulario de nuestro cliente y utilizaremos la opción **FILE—USE UNIT** (véase la Figura 14.14) para agregar una referencia a los dos módulos cliente DataSnap. Esto nos permitirá hacer referencia a los elementos enumerados previamente con

el objetivo de consumir el mismo servicio al que antes accedíamos mediante los componentes FireDAC.

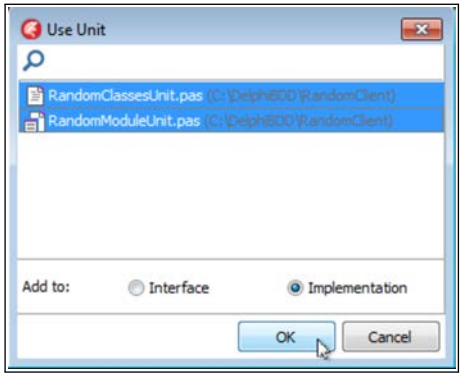


Figura 14.14 AGREGAMOS AL FORMULARIO REFERENCIAS A LOS MÓDULOS CLIENTE.

Aprovechando el formulario que ya teníamos, podemos agregar a la derecha un nuevo grupo de controles con la misma estructura del ya existente, a fin de facilitar la introducción de los límites y la invocación al servicio remoto. Hacemos doble clic sobre el nuevo botón e introducimos el siguiente código:

```
1 procedure TfrmMain.Button2Click(Sender: TObject);
2 begin
3   lblResultado2.Text := IntToStr(
4     RandomModule.RandomMethodsClient.NextRand(
5       StrToInt(edInferior2.Text),
6       StrToInt(edSuperior2.Text)
7     )
8   );
9 end;
```

Listado 14.5 Titulo

La variable `RandomModule` se define en el módulo que contiene el componente `TSQLConnection`, facilitando mediante la propiedad `RandomMethodsClient` una instancia de la 2ª clase, donde esta definido

el método `NextRand`. Lo usamos como si fuese local, mostrando el resultado en el `TLabel` correspondiente. En la Figura 14.15 puede verse el programa en funcionamiento, consumiendo el servicio mediante FireDAC y con el modulo cliente generado por el asistente.

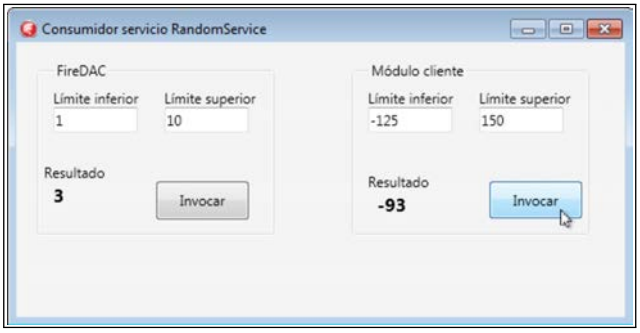


Figura 14.15 EL CLIENTE CON LAS DOS VÍAS PARA CONSUMIR EL SERVICIO

14.2.4 Control del estado del servidor

Al activarse nuestro actual servidor DataSnap, queda a la escucha en el puerto TCP/IP indicado, en cuanto el contenedor se ejecuta. En ocasiones puede interesar que esto no sea así, dejando el control del estado del servidor en manos de un hipotético administrador. Para ello usaríamos el formulario agregado por defecto al proyecto del servidor, actualmente vacío.

Supongamos que queremos incluir este tipo de funcionalidad en nuestro actual servidor. Para ello reproduciríamos los pasos indicados a continuación:

1. Abrimos el módulo contenedor del proyecto, en el que se encuentra el componente `TDSServer`, y damos el valor `False` a su propiedad `AutoStart`. De esta manera desactivamos el inicio automático del servidor al ejecutar el contenedor.

2. A continuación abrimos el formulario que forma parte del proyecto y que por ahora no habíamos usado. Introduciremos en el mismo un `TEdit`, que servirá para que el usuario indique el puerto donde quedará a la escucha

el servicio, y dos `TButton` que permitirán iniciar y detener el servidor. El `TEdit` y el botón de parada estarán inicialmente inaccesibles (propiedad `Enabled` a `False`).

3. El código asociado al evento `Onclick` de los dos botones será el mostrado a continuación:

```
...
procedure TForm10.btnIniciarClick(Sender: TObject);
begin
  ServerContainer1.DSTCPServerTransport1.Port := StrToInt(↙
      ↳ edPuerto.Text);
  ServerContainer1.DSServer1.Start;
  btnIniciar.Enabled := False;
  btnDetener.Enabled := True;
  edPuerto.Enabled := False;
end;

procedure TForm10.btnDetenerClick(Sender: TObject);
begin
  ServerContainer1.DSServer1.Stop;
  btnIniciar.Enabled := True;
  btnDetener.Enabled := False;
  edPuerto.Enabled := True;
end;
...
```

Listado 14.6 Código de los botones para iniciar y detener el servidor

Ahora, al ejecutar el proyecto, el contenedor se iniciará y hará visible el formulario de control. El servidor, sin embargo, no estará en funcionamiento, algo que podemos comprobar fácilmente desde el cliente de ejemplo desarrollado antes. Al intentar invocar al servicio (véase la Figura 14.16) se obtiene un error. Tendremos, por tanto, que iniciar antes el servicio para poder utilizarlo. Si cambiásemos el puerto tendríamos que tener en cuenta dos aspectos: que ese puerto esté disponible y que el cliente cambie su configuración para poder utilizarlo.

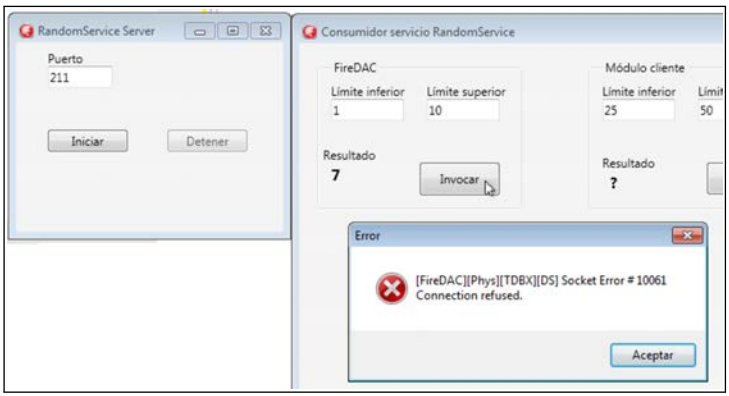


Figura 14.16 PODEMOS CONTROLAR EL ESTADO DEL SERVIDOR

14.3 DataSnap y conjuntos de datos

Los métodos que aparecen como servicios de un servidor DataSnap pueden devolver cualquier tipo de resultado, no solo datos simples como números enteros o cadenas de caracteres. Entre esos tipos potenciales se encuentran los conjuntos de datos, derivados de `TDataSet`, lo cual abre las puertas a la transferencia del resultado de una consulta desde el servidor al cliente, el tratamiento por parte del usuario en su dispositivo y la devolución de los cambios de vuelta al servidor, para su aplicación en el RDBMS.

En el pasado este trabajo se realizaba usando dbExpress, mediante componentes como `TDataSetProvider` en el servidor y `TDSProviderConnection` y `TClientDataSet` en el cliente. De hecho, prácticamente toda la documentación electrónica de Delphi sigue describiendo el procesamiento de conjuntos de datos remotos apoyándose en dichos elementos, ya que automatizan una buena parte del trabajo a realizar.

FireDAC es el mecanismo de acceso a datos preferente en las nuevas versiones de Delphi, razón por la que en este libro nos hemos centrado en su uso. Por ello en los siguientes apartados se describirá cómo tratar conjuntos de datos en aplicaciones DataSnap mediante FireDAC, en lugar de recurrir a dbExpress y los componentes que acaban de citarse.

14.3.1 Aspectos específicos en el servidor

Comenzaremos ocupándonos de los elementos que precisamos en el servidor de aplicaciones. El objetivo que este sea el que conecte con el RDBMS, facilitando los datos a la interfaz de usuario. Actuaría, por tanto, como intermediario entre el servidor de datos y el cliente. Esa disposición le permitiría efectuar tareas como la centralización del proceso de autenticación frente al RDBMS o la elaboración de ciertos resultados a partir de los datos antes de enviarlos a la interfaz.

Al crear el proyecto de un servidor DataSnap que va a ofrecer módulos de datos, en el paso del asistente mostrado en la Figura 14.17 elegiremos como ascendiente de la clase en la que se implementarán los métodos de servicio la clase `TDSServerModule` en lugar de `TComponent`.

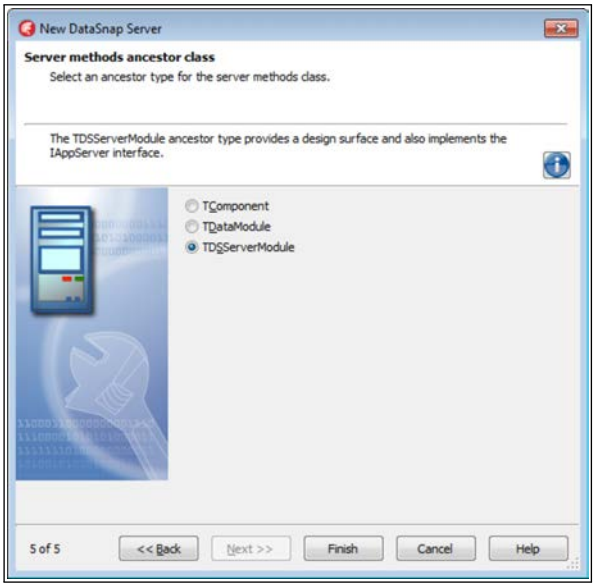


Figura 14.17 LA CLASE CON LOS MÉTODOS DE SERVICIO SERÁ DESCENDIENTE DE TDSSERVERMODULE

De esta forma la citada clase con los métodos de servidor tendrá asociado un contenedor, similar a un módulo de datos, que nos servirá para alojar los

componentes de acceso al RDBMS y configurar los conjuntos de datos que vayan a ofrecerse al cliente. Por otra parte, al derivar de `TDSServerModule` esta clase expone automáticamente un conjunto de métodos genéricos que el cliente podría utilizar para enumerar los proveedores de datos disponibles en el servidor, obtener filas de datos, transferir los cambios del cliente al servidor, etc.

> **NOTA**
>
> *Asumiendo que utilizaremos TCP/IP como protocolo de comunicación, es importante verificar que en la cláusula `uses` del módulo `ServerContainer` del servidor se ha agregado una referencia al módulo `IPPeerClient`. Análogamente, en el cliente habrá que introducir una referencia al módulo `IPPeerClient`. Estos módulos pertenecen a Indy, la biblioteca de comunicación subyacente usada por DataSnap.*

Suponiendo que deseamos tener en el servidor un mecanismo que nos permite ejecutar cualquier consulta sobre la base de datos `AdventureWorks`, usada en ejercicios de capítulos previos, agregaremos al módulo que contiene los métodos de servidor un `TFDConnection`, configurado para enlazar con la citada base de datos, y un `TFDQuery` enlazado al anterior. El módulo quedaría como puede apreciarse en la Figura 14.18.

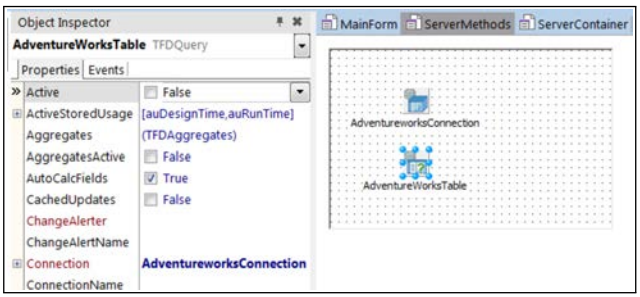


Figura 14.18 COMPONENTES EN EL MÓDULO DE CLASES DEL SERVIDOR DATASNAP

EJEMPLO 14.3 `DSCategories.dproj`

Puedes encontrar este proyecto completo en la carpeta `DSCategories`. Deberás contar con un servidor de datos SQL Server y la base de datos `AdventureWorks`. También puedes cambiar la configuración del componente `TFDConnection` para usar cualquier otra base de datos.

14.3.2 Devolución de un conjunto de datos

Una vez que se hayan añadido a este módulo los elementos necesarios, solamente tendremos que introducir en la clase un método de servicio por cada conjunto de datos a devolver. Cada método sería una función, opcionalmente con una lista de parámetros, que devolvería un dato de tipo `TDataSet`.

Podemos escribir un método específico para conjunto o subconjunto de datos, introduciendo en el método de servidor la consulta que habría que enviar al RDBMS. También existe la posibilidad de que el método acepte como argumento la consulta, la transfiera al RDBMS y devuelva el conjunto de datos resultante al cliente, sin más intervención. Esto último es lo que vamos a hacer en nuestro servidor. Primero agregaremos a la clase derivada de `TDSServerModule` la definición de un método que acepta como parámetro una cadena y devuelve como resultado un objeto `TDataSet`. A continuación introduciremos la implementacion de dicho método, que sería la mostrada a continuación:

```
1 function TCategoriesMethods.GetCategories(query: String): ↙
    ↳ TDataset;
2 begin
3   with AdventureWorksTable do begin
4     if Active then Close;
5     SQL.Clear;
6     SQL.Add(query);
7     Open;
8   end;
```

```
9
10    result := AdventureWorksTable;
11  end;
```

Listado 14.7 Método en el servidor para devolver el resultado de una consulta

Presta atención a la última sentencia del método. En ella se toma el componente `TFDQuery`, que es un derivado de la clase `TDataSet`, y se devuelve como resultado. Previamente se ha introducido en la propiedad `SQL` la consulta y se ha abierto el conjunto de datos, desencadenando así su ejecución.

14.3.3 Recepción de los datos en el cliente

La aplicación cliente, que será un proyecto de tipo multidispositivo, accederá a los datos ofrecidos por el servidor DataSnap de forma similar a como utilizaría cualquier otro origen de datos FireDAC. Podemos recurrir al Explorador de datos para localizar el método del servidor (véase la Figura 14.19) y arrastrarlo hasta el formulario, agregando automáticamente los componentes `TFDConnection` y `TFDStoredProc` adecuadamente configurados.

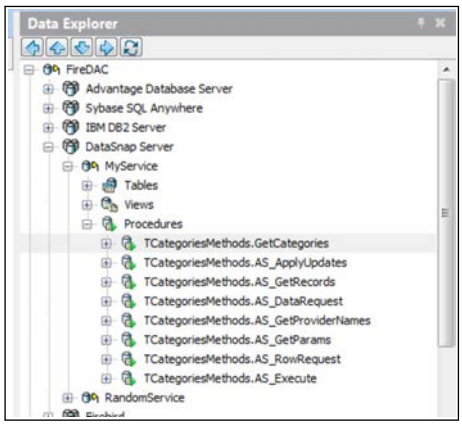


Figura 14.19 TOMAMOS EL PROCEDIMIENTO DEL EXPLORADOR DE DATOS

Una vez que tenemos el componente `TFDStoredProc` en el formulario[3], seleccionaremos su propiedad `Params` y abriremos el editor específico asociado. En este encontraremos los parámetros de entrada que acepte el procedimiento almacenado, en este caso solo uno, y el valor de retorno como último parámetro.

Seleccionamos el parámetro de entrada a fin de configurarlo en el Inspector de objetos. En este abrimos la propiedad `Value`, damos el valor `String` a la subpropiedad `Type` y escribimos la consulta que deseemos ejecutar, como se muestra en la Figura 14.20.

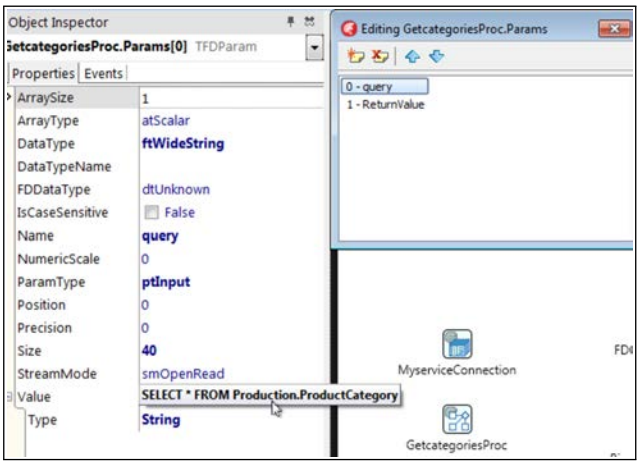


Figura 14.20 INTRODUCIMOS LA CONSULTA QUE QUEREMOS EJECUTAR

El siguiente paso consistiría en diseñar una interfaz de usuario básica para visualizar los datos. Con este fin podemos agregar al formulario un control `TGrid`, usando el diseñador visual de LiveBindings para enlazarlo con el componente `TFDStoredProc`, como se aprecia en la parte inferior de la Figura 14.21. En cuanto demos el valor `True` a la propiedad `Active` de dicho componente podremos ver los datos en el diseñador. Para ello es indispensable que el servidor DataSnap esté ejecutándose, como es lógico.

[3]Alternativamente podemos introducir los componentes `TFDConnection` y `TFDStoredProc` en un módulo de datos, como hemos hecho en los ejercicios de las dos partes previas del libro.

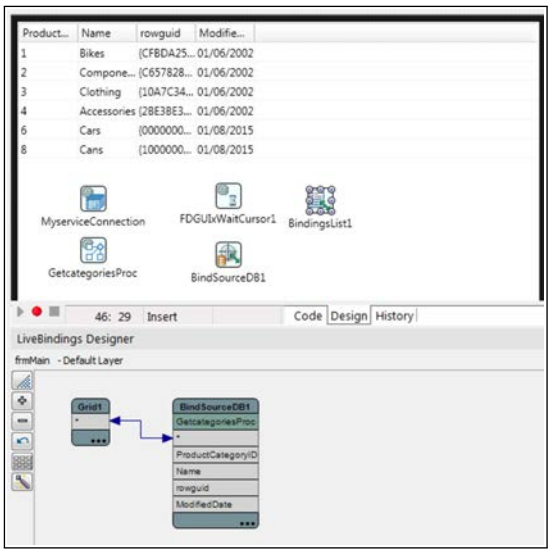


Figura 14.21 OBTENEMOS LOS DATOS DEL SERVIDOR EN EL DISEÑADOR

EJEMPLO 14.4 `DSCategoriesClient.dproj`

Puedes encontrar este proyecto completo en la carpeta `DSCategoriesClient`. Para que la aplicación funcione correctamente es necesario que esté en funcionamiento el servidor desarrollado en el paso previo.

Ahora ya podemos ejecutar el cliente y, siempre que el servidor esté también en funcionamiento y ambos puedan comunicarse, acceder a los datos remotos. Las tres capas de la solución: RDBMS, servidor DataSnap y aplicación multidispositivo, podrían estar ejecutándose cada una en una máquina distinta, incluso geográficamente distantes. Solo se necesitaría ajustar la dirección IP en la que se encuentra cada uno. Tenemos, por tanto, una aplicación distribuida, frente a la solución cliente/servidor del ejercicio anterior.

14.3.4 Devolución de los cambios

Para que el usuario pueda introducir cambios en los datos que ha recibido, teniendo en cuenta que su programa es una mera interfaz de usuario y no tiene conexión con el RDBMS, tendríamos que introducir los siguientes cambios en la pareja de proyectos previa:

- El componente `TFDStoredProc` usado en el cliente para ejecutar el método remoto y obtener el conjunto de datos resultante ha de configurarse de la siguiente forma:
  - Se ha de dar el valor `True` a la propiedad `CachedUpdates` a fin de activar el registro de cambios, evitando además que el componente intente comunicarse con la base de datos para transferirlos de forma inmediata.
  - En la propiedad `UpdateOptions.UpdateTableName` habrá que introducir el nombre de la tabla que se actualizará. En en este ejemplo el valor sería `Production.ProductCategory`. De esta forma se generarán las sentencias de actualización correspondientes.
- Será necesario introducir en el cliente un método que se encargue de enviar los cambios al servidor, ya sea a intervalos regulares, al cerrar la aplicación o bajo demanda del usuario. Para ello existen fundamentalmente dos alternativas:
  - Utilizar el método `AS_ApplyUpdates` expuesto automáticamente por la clase `TDSServerModule`, facilitando la lista de parámetros adecuada. Este método está diseñado para que su invocación sea sencilla usando dbExpress, por lo que alternativamente podríamos introducir los proveedores de datos (`TDataSetProvider`) y otros componentes necesarios para automatizar todos los pasos.
  - Implementar en el servidor un nuevo método al que se enviaría la lista de cambios a aplicar, invocándolo desde el cliente. Este puede obtener de la propiedad `Delta` del `TFDStoredProc` el registro de cambios, transferirlo al servidor y dejar que este lo asigne como datos de un `TFDQuery` o similar y efectúe la llamada a `ApplyUpdates`.

Lógicamente, este trabajo será necesario únicamente en caso de que se permita al usuario actuar directamente sobre todo el conjunto de datos. Si la in-

terfaz de usuario cuenta con apartados para agregar una nueva entrada, eliminarla, etc., estás acciones podrían implementarse en el servidor como métodos individuales a los que se invocaría desde el cliente con los correspondientes `TFDStoredProc`, facilitando la lista de parámetros necesaria en cada caso.

14.4 Resumen

Al finalizar este capítulo ya sabemos cómo utilizar DataSnap para crear servidores que ofrecen servicios a clientes remotos. Estos servidores pueden facilitar a dichos clientes operaciones diversas, implicando o no la devolución de conjuntos de datos. Dependiendo de ello tendremos una arquitectura cliente/servidor (dos capas) o distribuida en tres capas.

Hasta ahora hemos utilizado TCP/IP como medio para comunicar el servidor con los clientes y una representación interna propia para transferir los parámetros y datos entre ambas partes. El servidor DataSnap expone sus servicios como métodos a los que es posible invocar de forma remota. En el capítulo siguiente veremos cómo construir con DataSnap servidores de aplicaciones que atienden peticiones REST y usan JSON como representación de la información a transferir.

Capítulo 15

SERVICIOS REST

En el capítulo 13 se introdujeron, entre otros aspectos, las bases de REST como mecanismo para facilitar el acceso a recursos remotos, mediante el protocolo de comunicación HTTP. Asimismo se explicó cómo usar JSON como formato para expresar los datos que viajarán entre clientes y servidor de aplicaciones, describiendo algunos servicios de Delphi para construir documentos JSON e interpretarlos, extrayendo los datos que contienen.

Partiendo de ese conocimiento, en el presente capítulo aprenderemos a construir servidores de aplicaciones DataSnap de tipo REST. Los principios REST, HTTP y JSON son los pilares fundamentales de este tipo de servidores, mediante

los cuales se extienden las capacidades de DataSnap descritas en el capítulo previo. Al igual que en dicho capítulo, comenzaremos desarrollando un servidor básico para a continuación pasar a operar con conjuntos de datos.

También se introducen los componentes que nos permitirán consumir desde una aplicación Delphi cualquier servicio REST. Además de los clientes desarrollados con Delphi, aprenderemos asimismo a acceder a estos servidores DataSnap desde una página web, un cliente ligero que puede ejecutarse en cualquier tipo de dispositivo, incluyendo móviles.

15.1 El asistente DataSnap REST

Comenzaremos examinando los pasos del asistente que incorpora Delphi para la generación de proyectos de tipo DataSnap REST, especialmente las diferencias existente con el asistente que conocimos en el capítulo previo. Para iniciar el asistente abriremos el cuadro de diálogo **NEW ITEMS**, concretamente la rama **DATASNAP SERVER**, y elegiremos la opción **DATASNAP REST APPLICATION**.

Como ya sabemos, el número de pasos del asistente es variable dependiendo de las opciones que van eligiéndose. Los enumerados a continuación serian los correspondientes para un servidor que tiene como contenedor una aplicación estándar FMX y no usa HTTPS:

1. En el primer paso podremos elegir como tipo de contenedor, además de aplicaciones de consola y con GUI, módulos de servidor web para IIS y Apache (véase la Figura 15.1)
2. Al seleccionar la opción **STAND-ALONE APPLICATION** en el paso previo, en este podremos elegir entre las bibliotecas FMX y VCL.
3. En el tercer paso podremos elegir el puerto HTTP en el que el servidor estará a la escucha. Por defecto se usa el puerto `8080`. También se ofrece la opción para activar la comunicación por HTTPS, para lo cual precisaremos el certificado correspondiente.
4. En este paso podremos elegir los elementos que queremos incluir en el proyecto. Además de los que conocimos en el capítulo previo, como el módulo

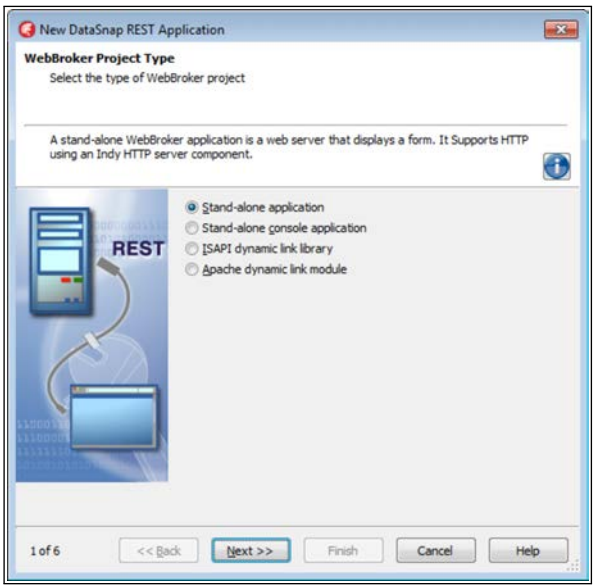


Figura 15.1 SELECCIONAMOS EL TIPO DE APLICACIÓN CONTENEDORA DEL SERVIDOR

con la clase que expone los módulos de servicio o los componentes de autenticación, hay otros adicionales (véase la Figura 15.2), como los archivos web de ejemplo para invocar a uno de los métodos del servidor.

5. A continuación indicaremos si la clase de métodos del servidor ha de derivarse de `TComponent` o de `TDataModule`. Como ya sabemos, esto último ofrece un contenedor en el que alojar componentes de acceso a datos.

6. Dependiendo de los elementos marcados en el cuarto paso, en este se nos pedirá que facilitemos una ruta en la que almacenar los archivos HTML, CSS y JavaScript de ejemplo.

Como podemos comprobar, en este caso no existe la opción de utilizar TCP/IP para comunicarse directamente con el servidor. Las peticiones siempre se efectuarán mediante HTTP, ajustándonos a los principios REST. Por defecto todos los componentes se incluirán en un módulo web, una clase derivada de

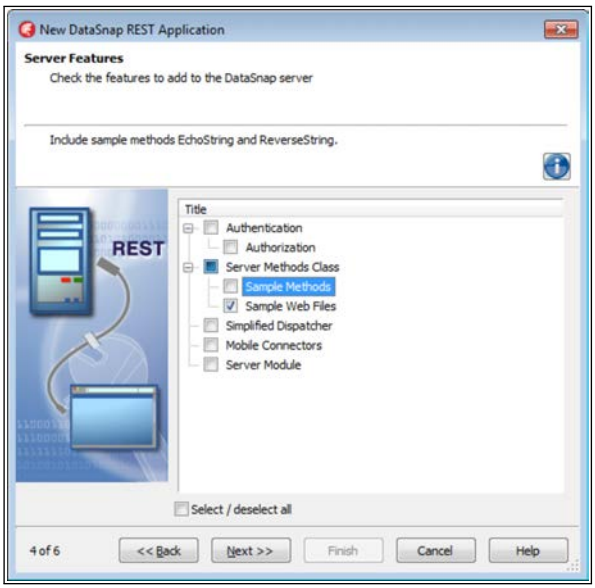


Figura 15.2 ELEGIMOS LOS ELEMENTOS A INCLUIR EN EL PROYECTO

`TWebModule` que hace innecesario el módulo de servidor[1]. Si el asistente tiene un formulario (se ha elegido la primera opción en el paso 1), este incluirá los elementos precisos para controlar el estado del servidor. En los siguientes apartados se enumera con más detalle el contenido de cada módulo.

15.1.1 Módulo de clases del servidor

Este módulo es el más simple, especialmente si no hemos marcado la opción **SIMPLE METHODS** en el cuarto paso del asistente. Solo contendrá la definición de una clase vacía. Esta estará enmarcada entre las directivas `$METHODINFO ON` y `$METHODINFO OFF`, cuya finalidad se explicó en el capítulo previo.

[1]Este, no obstante, puede agregarse al proyecto marcando la opción correspondiente en el cuarto paso del asistente.

En teoría, para ajustarnos a los principios REST, el nombre de cada método incluido en la clase de servicio debería identificar a un recurso, devolviéndolo como resultado. Por cada recurso podemos implementar en la clase cuatro métodos distintos, asociados a los siguientes tipos de solicitud HTTP:

- `GET`: Una solicitud de este tipo ejecutaría el método base que identifica el nombre del recurso. Se espera que dicho método, como acaba de indicarse, devuelva como resultado dicho recurso. Por regla general este será una colección de entradas de algún tipo.
- `POST`: Con este tipo de solicitud se procedería a ejecutar el método llamado `updateRecurso`, siendo `Recurso` el nombre del método base. Su finalidad será actualizar algún dato devuelto por el método previo, por ejemplo modificando uno de las entradas de la colección.
- `PUT`: La finalidad de este tipo de solicitud es agregar un nuevo valor a la colección de entradas devuelta por el método base. El nombre del método a ejecutar sería `acceptRecurso`.
- `DELETE`: En este caso el método a ejecutar sería `cancelRecurso`, siendo su objetivo eliminar una de las entradas de la colección obtenida con `GET`.

NOTA

No es un requerimiento que para cada método base se implementen los tres métodos adicionales de actualización, inserción y eliminación. Los incluiremos o no dependiendo de la funcionalidad específica de nuestro servidor.

15.1.2 Módulo web

La clase definida en este módulo, derivada de `TWebModule`, incorpora por sí misma la funcionalidad necesaria para responder a solicitudes HTTP redirigiendo estas a los componentes que correspondan. Dichos componentes se alojarán en el módulo web, cuya apariencia en el diseñador es similar a la de un módulo de datos.

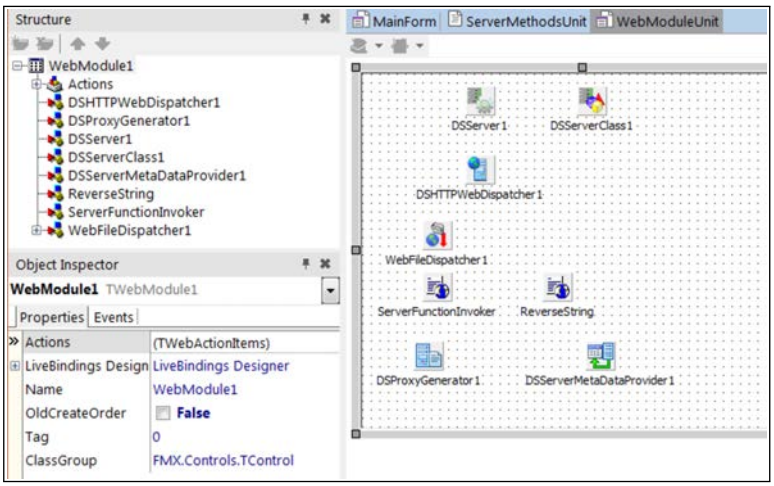


Figura 15.3 COMPONENTES AGREGADOS POR EL ASISTENTE AL MÓDULO WEB

El módulo web generado por el asistente contendrá los elementos que pueden verse en la Figura 15.3. La finalidad de cada uno de ellos es la indicada a continuación:

- `TDSServer` y `TDSServerClass`: Estos dos componentes ya los conocemos del capítulo previo. Su finalidad es la que se explicó en su momento y son esenciales para cualquier servidor DataSnap, ya sea de tipo REST o clásico.
- `TDSHTTPWebDispatcher`: Es el componente encargado de gestionar las solicitudes REST provenientes de los clientes. Establece, entre otros parámetros, los tipos de solicitudes HTTP a las que se responderá y varios componentes de la ruta de acceso a los servicios. En el siguiente apartado se ofrecen más detalles sobre la configuración de este componente.
- `TWebFileDispatcher`: La finalidad de este componente es responder a la solicitud de archivos estáticos alojados en el servidor. Esto son habitualmente imágenes, hojas de estilo y módulos JavaScript a los que se hace referencia desde páginas HTML. Este componente cuenta con dos propiedades que hemos de configurar:

- `WebDirectories`: Contiene una colección de directorios indicando si son accesibles o no. Cuenta con un editor específico con el que podemos agregar entradas.
- `WebFileExtensions`: Al igual que la anterior, esta también cuenta con un editor específico (véase la Figura 15.4) que permite configurar las extensiones de archivo para las que se aceptan solicitudes. Cada entrada establece la extensión y el tipo MIME que le corresponde.

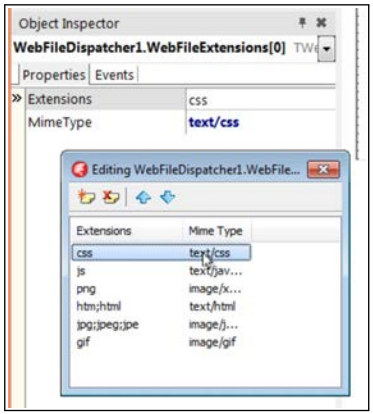


Figura 15.4 CONFIGURACIÓN DE LAS EXTENSIONES DE ARCHIVOS ACEPTADAS

- `TPageProducer`: Cuando se quiere servir una página web introduciendo contenido dinámico, generado por código Delphi en el momento de la solicitud, se recurre a este componente. El asistente introduce dos en el proyecto creado: una que permite acceder a cualquier función implementada por el servidor y ejecutarla y otro que entrega una página de ejemplo para consumo de un servicio específico. El contenido a devolver se facilita en la propiedad `HTMLDoc` o bien entregando la ruta y nombre de un archivo en la propiedad `HTMLFile`. Al recibir la solicitud para el contenido representado por el `TPageProducer`, este analiza el contenido y genera un evento `OnHTMLTag` cada vez que encuentra una marca del tipo `<#nombre>`, facilitando la sustitución por el contenido dinámico que interese.
- `TDSServerMetaDataProvider` y `TDSProxyGenerator`: Estos dos componentes trabajan conjuntamente a fin de recuperar información sobre

los métodos disponibles en el servidor, el primero, y generar el módulo *proxy*[2] correspondiente a fin de facilitar el acceso a los mismos, el segundo. El primero se conecta al servidor DataSnap en ejecución de forma directa, están en el mismo proceso, y recupera la información que necesita el segundo para realizar su trabajo. Podría utilizarse un `TDSRestMetaDataProvider` para conectar con un servicio REST remoto mediante un componente `TDSRestConnection`. Con la información obtenida del anterior, el `TDSProxyGenerator` genera el *proxy* en el lenguaje indicado por la propiedad `Writer`. Como se aprecia en la Figura 15.5 esta tiene asociada una lista desplegable de la que podemos elegir la configuración de salida.

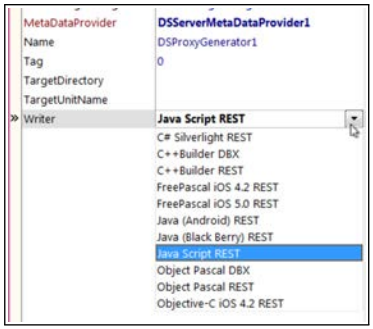


Figura 15.5 SELECCIÓN DEL TIPO DE *proxy* A GENERAR

Teniendo una visión general de cuál es el cometido de cada uno de los componentes existentes en el módulo web, veamos ahora con más detalle la configuración de algunos aspectos importantes del mismo.

Configuración de acciones web

El módulo web, en el que están alojados los componentes que acaban de enumerarse, es el responsable de redirigir las solicitudes entrantes al punto adecuado.

[2]Un *proxy* DataSnap es un módulo con código a medida para facilitar la invocación de los servicios ofrecidos por el servidor desde clientes escritos en distintos lenguajes de programación.

Para ello se define una colección de objetos `TWebActionItem`, cada uno de los cuales vincula una ruta de acceso al servidor con el contenido o productor de contenido correspondiente. Podemos abrir el editor de dicha colección con el botón asociado a la propiedad `Actions`.

Mediante el editor asociado a dicha propiedad, situado en la parte inferior de la Figura 15.6, podemos agregar, eliminar y modificar el orden de las acciones. En la parte superior puede verse que podemos configurar el tipo de solicitud al que responde cada acción, su nombre, ruta y el enlace al componente que producirá el contenido que, en esta imagen, es uno de los `TPageProducer`.

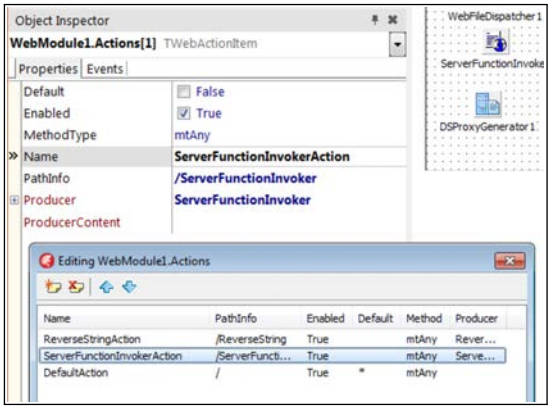


Figura 15.6 EDICIÓN DE LAS ACCIONES ASOCIADAS AL MÓDULO WEB

> **NOTA**
>
> *No es necesario definir acciones para el acceso a los servicios individuales mediante las URL de tipo* `http://servidor/clase/método`.

Configuración de las solicitudes a atender

Uno de los aspectos fundamentales al desarrollar un servidor DataSnap REST es la configuración de las solicitudes que se atenderán, incluyendo las partes de la

ruta que de acceso a los servicios. Este trabajo queda en manos del componente `TDSHTTPWebDispatcher`.

El URL de acceso a un servicio ofrecido por un servidor DataSnap REST se compondrá de las siguientes partes:

```
http://servidor.dom/DSContext/RESTContext/Clase/Método/Arg
```

En caso de que el `TDSHTTPWebDispatcher` no estuviese conectado a un componente `TDSServer` mediante la propiedad `Server`, lo cual no es habitual en un proyecto DataSnap, el nombre del servidor se indicaría en la propiedad `DSHostName`. El resto de componentes de la ruta se configuran de la siguiente manera:

- `DSContext`: Es la parte de la ruta que corresponde al contexto DataSnap. Lo establece la propiedad `DSContext` y por defecto su valor es `datasnap/`.
- `RESTContext`: Indica el contexto REST en el servidor. El valor de esta parte de la ruta por defecto es `rest/`. Podemos cambiarlo modificando el valor de la propiedad `RESTContext`.
- `Clase`: Será el nombre de la clase en la que se implementan los métodos de servicio, tal y como aparece en el módulo de definición de clases del servidor.
- `Método`: Esta parte de la ruta indicará el nombre base del método al que se quiere acceder. El tipo de solicitud HTTP determinará qué método sea el invocado, según se explicó antes en la Sección 15.1.1.
- `Arg`: Tras el nombre del método se entregará cualquier argumento adicional que este pudiera necesitar. Si hay más de un parámetro, cada uno se separará del anterior con una barra.

Además de los componentes de la ruta, que podemos ajustar cambiando las propiedades `DSContext` y `RESTContext` y el nombre de la clase y los métodos, el componente `TDSHTTPWebDispatcher` también determinar qué solicitudes REST se atenderán. Con este fin se utiliza la propiedad `WebDispatch` que, como se aprecia en la Figura 15.7, está compuesta de tres subpropiedades. Con la propiedad `MethodType` se establecen los tipos de solicitudes a atender. Por

defecto su valor es `mtAny`, por lo que se atenderían todas. Podemos, no obstante, modificar su valor para restringirlas. La propiedad `PathInfo` fija la ruta a la que han de llegar las solicitudes a atender. Por defecto es `datasnap*`. Esto implica que el servidor responda a todas aquellas solicitudes que lleguen a `http://servidor/datasnap/XXX`. El valor de esta propiedad debe coincidir con el asignado a la propiedad `DSContext`, agregando el carácter `*` como sufijo.

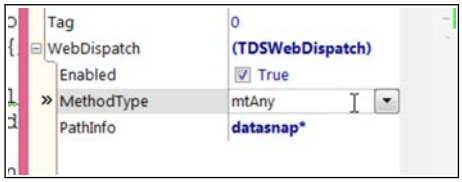


Figura 15.7 CONFIGURAMOS LAS PETICIONES QUE SE ATENDERÁN.

15.1.3 Módulo de interfaz

El último módulo que nos queda por examinar es el correspondiente a la interfaz de usuario, asumiendo que hemos elegido esta opción en el asistente descrito con anterioridad. El formulario, al contrario de lo que ocurría con los servidores DataSnap desarrollados en el capítulo previo, no está vacío. En él encontramos (véase la Figura 15.8) dos botones para iniciar y detener el servidor, un apartado para cambiar el puerto en que está a la escucha y un tercer botón para abrir la página inicial del servidor en el navegador por defecto.

Examinando el código incluido en el mismo módulo, en especial el asociado al evento `OnCreate` y la pulsación de los dos botones de control de estado, podremos comprobar lo fácil que es iniciar o detener el servidor.

15.2 Un servicio REST simple

Asumiendo que hemos creado un nuevo proyecto con las opciones por defecto del asistente, podemos completar la definición de este módulo agregando un

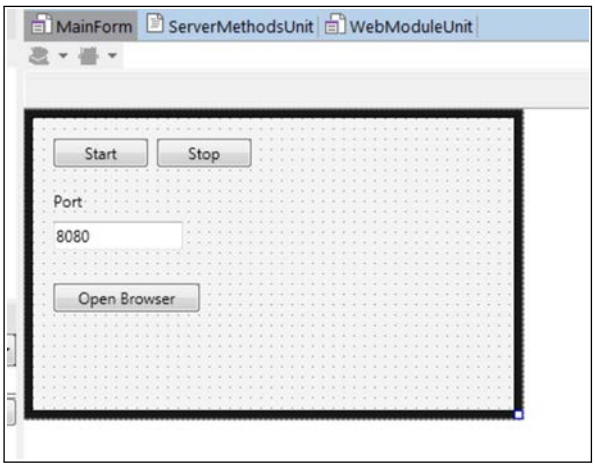


Figura 15.8 FORMULARIO DE CONTROL DEL SERVIDOR DATASNAP REST

método similar que definíamos en el primer ejercicio del capítulo previo, aunque en este caso lo llamaríamos simplemente `Rand`. Así tendremos un servicio que facilitará números aleatorios en un rango concreto. Dicho método respondería a peticiones de tipo `GET`. Admitiremos también solicitudes de tipo `POST` a fin de actualizar la semilla del generador de números aleatorios. El nombre del método encargado de esta tareas sería `updateRand` según el patrón descrito con anterioridad.

EJEMPLO 15.1 `RESTRandomService.dproj`

Puedes encontrar este proyecto completo en la carpeta `RESTRandomService`. Para ejecutarlo puede ser necesario abrir el puerto TCP/IP en el que el servidor queda a la escucha. Por defecto es el 8080 salvo que antes de iniciar el servidor lo cambios en el formulario de control.

Abrimos el módulo correspondiente a la clase que expone los métodos del servidor y modificamos la definición de la clase, dejándola como se muestra a continuación:

```
...
type
{$METHODINFO ON}
  TRandomMethods = class(TComponent)
  public
    function Rand(FromN: integer; ToN: integer): integer;
    function updateRand(seed: integer): integer;
  end;
{$METHODINFO OFF}
```

Listado 15.1 Definición de la clase con los dos métodos

A continuación introduciremos en la sección correspondiente la implementación de los dos métodos, que sería la siguiente:

```
implementation

function TRandomMethods.Rand(FromN, ToN: integer): integer;
begin
  Result :=  Random(ToN-FromN) + FromN;
end;

function TRandomMethods.updateRand(seed: integer): integer;
begin
  result := RandSeed;
  RandSeed := seed;
end;
```

Listado 15.2 Implementación de los métodos de servicio

Con esto ya tenemos finalizado nuestro primer servidor de tipo REST. Veamos ahora cómo podemos acceder a estos servicios directamente desde el navegador, lo cual nos permite comprobar su funcionamiento sin necesidad de crear un cliente a medida.

15.2.1 Comprobación del servidor desde el navegador

A fin de verificar que el servidor funciona adecuadamente, y podemos usar los métodos que ofrece, seguiremos el procedimiento indicado a continuación:

1. Ejecutamos el proyecto directamente desde el entorno de Delphi, esperando a que aparezca el formulario con la interfaz de control.

2. Hacemos clic en el botón **START** para poner el servidor en marcha.

3. Hacemos clic en el botón **OPEN BROWSER** para abrir en el navegador por defecto la página de inicio.

4. Abrimos una nueva pestaña en el navegador e introducimos el URL que puede verse en la parte superior de la Figura 15.9. Este funcionará siempre que no hayamos modificado las propiedades `DSContext` y `RESTContext` del componente `TDSHTTPWebDispatcher` ni modificado el nombre de la clase que contiene los métodos. Los dos números que aparecen al final de la ruta son los parámetros: límite inferior y superior del rango en el que estará el número devuelto.

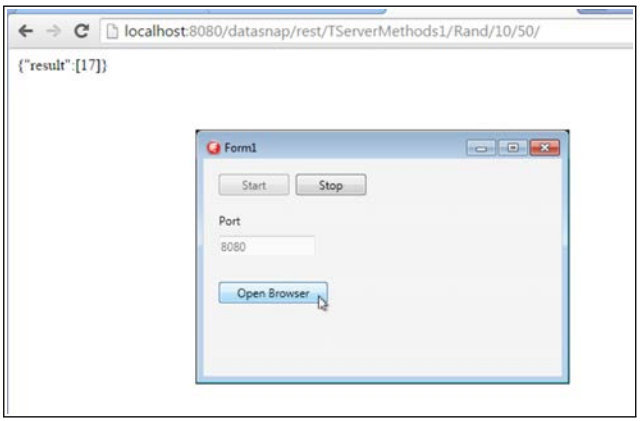


Figura 15.9 LLAMADA AL SERVICIO DESDE EL NAVEGADOR CON PETICIÓN GET

5. Cada vez que actualicemos la página deberíamos obtener un valor de retorno distinto. El resultado mostrado en el navegador corresponde al documento JSON devuelto por el servidor.

6. Para poder invocar al método que cambia la semilla del generador de números aleatorios no podemos introducir simplemente un URL, ya que la solicitud no es de tipo `GET` sino `POST`. Los parámetros de entrada, además, se facilitan en el cuerpo del mensaje, no en la URL. Por esta razón volvemos a la página inicial del servidor y hacemos clic en el vínculo **SERVER FUNCTIONS**.

7. Se abre una página con las distintas clases que exponen servicios accesibles para los clientes. Desplegamos la lista que corresponde a nuestra clase. Ahí deberían aparecer los dos métodos que ofrece (véase la Figura 15.10).

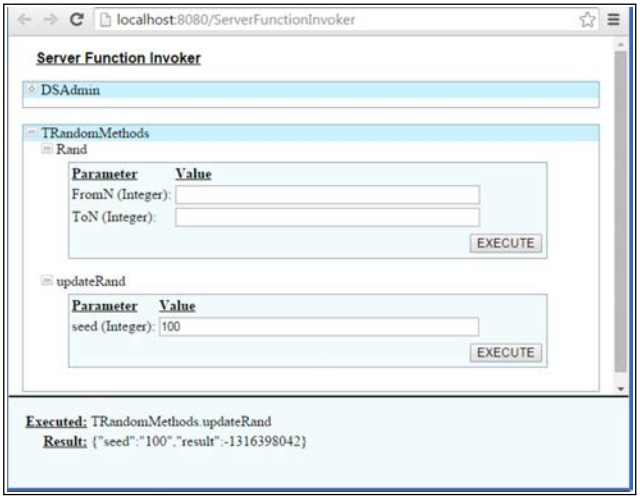


Figura 15.10 LLAMADA DE TIPO `POST` CON EL FORMULARIO DE INVOCACIÓN

8. Introducimos la nueva semilla y hacemos clic en el botón **EXECUTE** para enviar la solicitud. Cada vez que fijemos una semilla concreta el generador devolverá la misma secuencia de valores.

Si los resultados se ajustan a lo explicado sabremos que el servidor está funcionando como se espera. Podemos detenerlo, para liberar la interfaz de Delphi, y ejecutarlo por separado a fin de acceder al mismo desde otro tipo de clientes.

15.3 Acceso a servicios REST desde Delphi

Una vez que tenemos en funcionamiento un servicio que sigue los principios REST llega el momento de aprender a consumir este tipo de servicios desde aplicaciones Delphi. Para ello usaremos un conjunto de componentes que nos servirán para acceder a cualquier servicio, no necesariamente desarrollado con Delphi.

Delphi incorpora una herramienta, llamada **RESTDEBUGGER**, que facilita la depuración de servicios REST. Asimismo simplifica la configuración de los componentes que usaríamos en Delphi para consumirlo, tal y como se explicará en un apartado posterior.

15.3.1 Componentes de un cliente REST

Los componentes esenciales para crear un cliente REST, que podemos usar en proyecto de tipo multidispositivo, son los descritos a continuación:

- `TRESTClient`: Es el componente encargado de ponerse en contacto con el servidor, transferir la solicitud y recoger la respuesta, ocupándose de aspectos como la autenticación, pasar por el servidor *proxy* si existiese y otras tareas de comunicación.
- `TRESTRequest`: Aporta al componente anterior la información necesaria para componer la solicitud, indicando el tipo de petición HTTP, el recurso al que se quiere acceder y los parámetros asociados.
- `TRESTResponse`: Será el componente encargado de alojar la información obtenida como respuesta al procesar la solicitud, facilitando el acceso a la misma por parte de nuestra aplicación.

Estos tres componentes han de estar vinculados entre sí. Para ello el componente `TRESTRequest` dispone de las propiedades `Client`, que lo vincula con el `TRESTClient`, y `Response`, que lo enlaza con el `TRESTResponse`.

El procedimiento general de configuración de estos componentes a fin de acceder a un servicio REST cualquiera es el esquematizado a continuación:

1. Se facilita en la propiedad `BaseURL` del `TRESTClient` el URL base de acceso al servidor REST al que se quiere acceder. Opcionalmente se pueden configurar parámetros como el tipo de codificación, los tipos de contenido aceptados como respuesta, los datos de configuración del servidor *proxy*, etc.

2. Con la propiedad `Method` del `TRESTRequest` se establece el tipo de solicitud HTTP a usar para enviar la solicitud. El recurso a solicitar se especificará en la propiedad `Resource`, facilitando el nombre de los parámetros en caso de existir. Los valores para estos se introducirán en la colección asociada a la propiedad `Params`.

3. Se procederá a enviar la solicitud al servidor, usando para ello el método `Execute` del componente `TRESTRequest`.

4. Recibida la respuesta, se utilizarán algunas de las propiedades del componente `TRESTResponse` para obtener los datos devueltos por el servicio. Algunas de las propiedades de interés son `ContentType`, que especifica el tipo de contenido que se ha obtenido; `ContentLenght`, que indica el tamaño de los datos recuperados; `Content`, encargada de alojar el contenido, y `JSONValue`, que facilita los datos en formato JSON.

> **NOTA**
>
> *El menú contextual asociado al componente `TRESTRequest` cuenta con una opción* **EXECUTE** *que permite enviar la solicitud en fase de diseño. Esto nos permite probar la configuración sin necesidad de compilar ni ejecutar el proyecto.*

15.3.2 Consumo del servicio simple desde un cliente Delphi

Conociendo de forma general los componentes que debemos utilizar, nuestro siguiente paso será utilizarlos para consumir el servicio REST implementado por el servidor DataSnap desarrollado anteriormente. Se asume que dicho servidor está en ejecución en la misma máquina donde desarrollamos el cliente.

El proyecto cliente será una aplicación multidispositivo. Agregaremos al formulario los tres componentes antes citados y los configuraremos tal y como se indica a continuación:

- `TRESTClient`: Asignaremos a la propiedad `BaseURL` de este componente el valor `http://localhost:8080/datasnap/rest/TRandomMethods`. Este es el URL base de acceso al servidor. Las demás propiedades pueden dejarse con sus valores por defecto. Llamamos a este componente `RESTRand`

- `TRESTRequest`: Le asignaremos el nombre `RESTRandRequest`. Debemos modificar su propiedad `Resource`, a la que asignaremos el valor `Rand/FromN/ToN`. Al introducir entre llaves los nombres de los parámetros, estos se agregarán automáticamente a la colección `Params`. Abrimos el editor asociado a la misma, seleccionamos el parámetro `FromTo` y cambiamos el contenido de su propiedad `Value`, haciendo lo mismo con el segundo parámetro. De esta forma contamos con unos valores por defecto.

- `TRESTResponse`: En principio no tenemos que modificar nada en la configuración de este componente. Solamente nos aseguraremos de que está vinculado con el anterior, usando para ello la propiedad `Response` del `TRESTRequest`.

Aunque aún no hemos introducido en el formulario elemento alguno de interfaz que permita al usuario interactuar con el servicio, podemos probar el funcionamiento de este de forma inmediata. No tenemos más que abrir el menú contextual del componente `TRESTRequest` y elegir la opción **EXECUTE**, tal y como se aprecia en la Figura 15.11. En un pequeño cuadro de diálogo se mostrará el código de respuesta obtenido del servicio. Habitualmente obtendremos un código 200 si todo ha ido bien o uno distinto si se produce algún tipo de fallo.

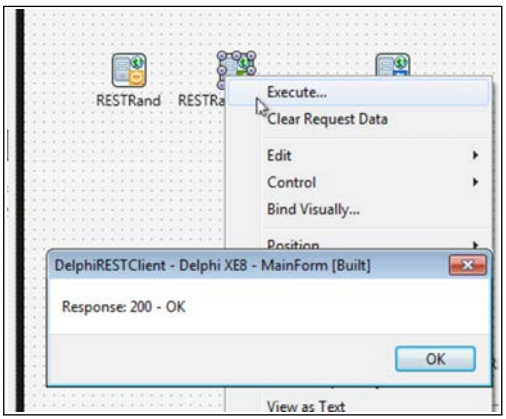


Figura 15.11 EJECUTAMOS LA SOLICITUD DESDE EL DISEÑADOR DE DELPHI

Tras el paso previo, podemos seleccionar el componente `TRESTResponse` para observar en el Inspector de objetos qué información se ha obtenido. La propiedad `ContentLength` indicará su tamaño. Abriendo el editor asociado a la propiedad `Content` podremos ver la respuesta en sí, en este caso el documento JSON con el número aleatorio (véase la Figura 15.12).

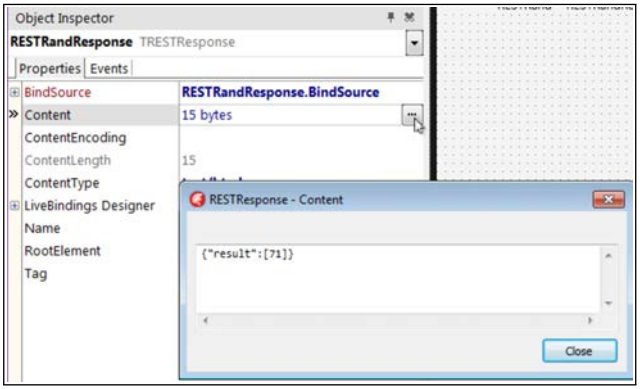


Figura 15.12 EXAMINAMOS LA RESPUESTA OBTENIDA DEL SERVICIO

EJEMPLO 15.2 `DelphiRESTClient.dproj`

Puedes encontrar este proyecto completo en la carpeta `DelphiRESTClient`. Para que la aplicación funcione correctamente es necesario que esté en funcionamiento el servidor desarrollado en el paso previo. En caso necesario ajusta la ruta de acceso asignada a la propiedad `BaseURL` del componente `TRESTClient` según la configuración de tu equipo de desarrollo.

A fin de poder acceder también a la función que cambia la semilla del generador de números aleatorios, segundo método del servidor, agregaremos al formulario un segundo `TRESTRequest` y un segundo `TRESTResponse`. Enlazaremos el primero con el `TRESTClient` ya existente y con el segundo, usando las propiedades `Client` y `Response`. La propiedad `Resource` de este segundo `TRESTRequest` contendrá el valor `Rand/seed`. Abrimos el editor asociado a la propiedad `Params` e introducimos un valor por defecto para el parámetro `seed`. Cambiamos el valor de la propiedad `Method` para efectuar una solicitud de tipo `POST` en lugar de `GET`. Podemos proceder como en el caso anterior, probando el funcionamiento de este método desde el diseñador.

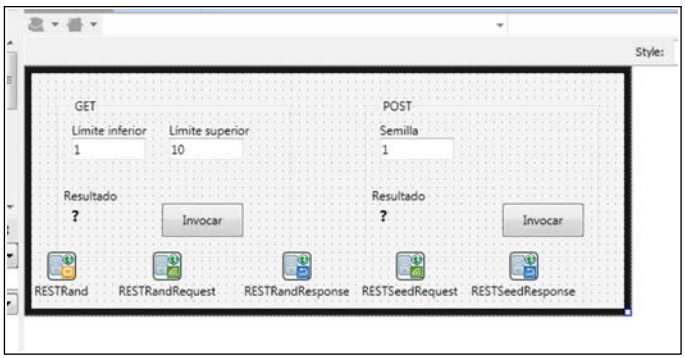


Figura 15.13 INTERFAZ DE USUARIO DEL CLIENTE REST DELPHI

El siguiente paso será diseñar una interfaz de usuario simple que permita al usuario introducir los parámetros e invocar al método a demanda. Los elementos de dicha interfaz serán los que aparecen en la Figura 15.13: dos `TGroupBox`, dos `TButton` para invocar a los servicios, dos `TLabel` para mostrar resultados y los `TEdit` precisos para introducir los valores de entrada. El código asociado a los dos botones es el mostrado a continuación:

```
1  procedure TfrmMain.btnRandClick(Sender: TObject);
2  begin
3    RESTRandRequest.Params[0].Value := edInferior.Text;
4    RESTRandRequest.Params[1].Value := edSuperior.Text;
5    RESTRandRequest.Execute;
6    lblResultado.Text := ((RESTRandResponse.JSONValue as ↙
         ↳ TJSONObject).Values['result'] as TJSONArray).Items↙
         ↳ [0].ToString;
7  end;
8
9  procedure TfrmMain.btnSeedClick(Sender: TObject);
10 begin
11   RESTSeedRequest.Params[0].Value := edSemilla.Text;
12   RESTSeedRequest.Execute;
13   lblResultSeed.Text := ((RESTSeedResponse.JSONValue as ↙
         ↳ TJSONObject).Values['result'] as TJSONArray).Items↙
         ↳ [0].ToString;
14 end;
```

Listado 15.3 Código para invocar a los servicios con los parámetros introducidos

Para extraer el valor que nos interesa del documento JSON obtenido como respuesta usamos los tipos y métodos descritos en el capítulo 13. Ahora ya podemos ejecutar el proyecto y llamar a los servicios ofrecidos por nuestro servidor REST. El funcionamiento es análogo al que obteníamos desde el navegador web.

15.3.3 El depurador REST

Delphi incorpora una herramienta que nos interesa conocer. Su finalidad es facilitar la prueba de servicios REST, así como simplificar la configuración de los

componentes necesarios para acceder a ellos. El depurador REST se inicia con la opción **TOOLS—RESTDEBUGGER**. Como se aprecia en la Figura 15.14, en la parte superior de la interfaz de usuario existen cuatro páginas. A su derecha están los botones que permiten enviar la solicitud, así como almacenar y recuperar configuraciones de peticiones. En la parte inferior podemos examinar las cabeceras, el cuerpo y los datos obtenidos como respuesta.

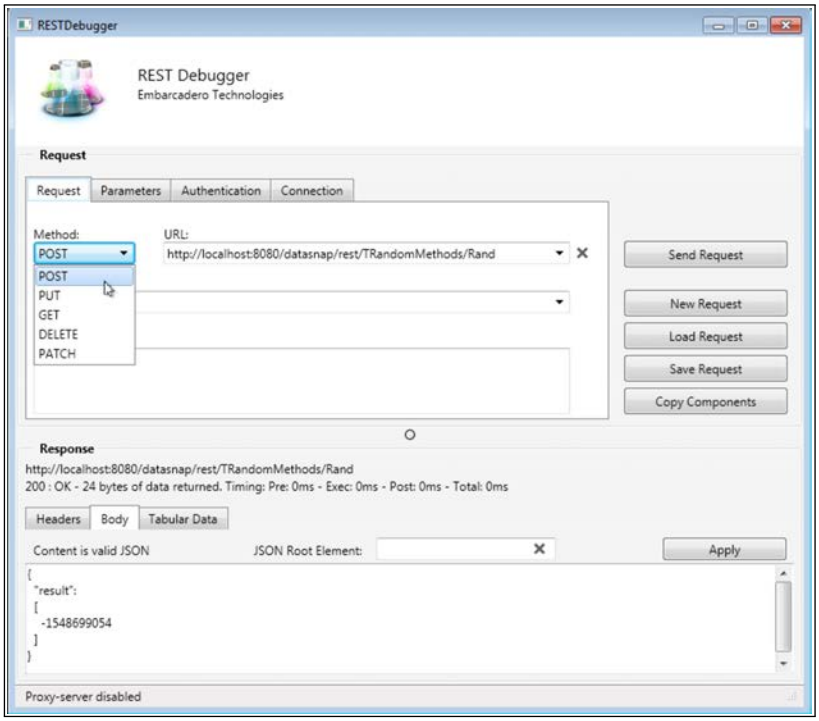


Figura 15.14 CONFIGURACIÓN DE LA PETICIÓN PARA CONSUMIR EL SERVICIO

Suponiendo que quisiésemos comprobar el funcionamiento de nuestro servicio de generación de números aleatorios con esta herramienta, los pasos básicos serían los siguientes:

1. En la página **REQUEST** Seleccionamos de la lista desplegable **METHOD** el método `GET` o `POST`, según deseemos obtener un número aleatorio o

cambiar la semilla, respectivamente. Se contempla el uso de otros tipos de solicitudes, aunque nuestro actual servidor no las soporta.

2. Introducimos en el apartado **URL** de la misma página el URL de acceso al servicio. Esta sería la misma para invocar a ambos métodos del servicio.
3. A continuación abriríamos la página **PARAMETERS** y, usando el botón **ADD**, agregaríamos los parámetros necesarios, como puede verse en la Figura 15.15.
4. Finalmente haríamos clic en el botón **SEND REQUEST** para enviar la solicitud y obtener la respuesta, que se mostraría en la parte inferior.

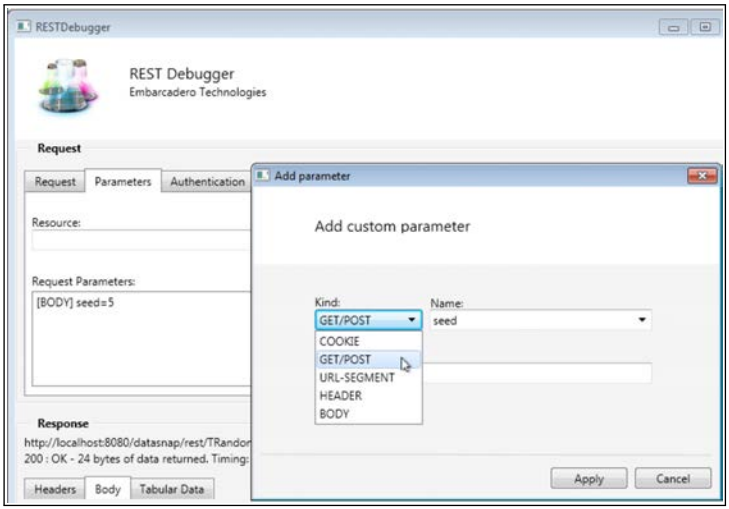


Figura 15.15 SE FACILITAN LOS PARÁMETROS QUE PRECISA EL SERVICIO

Además de permitirnos comprobar si un servicio funciona como debería, enviando solicitudes y examinando la respuesta obtenida, esta herramienta también nos permite obtener el trío de componentes `TRESTClient`, `TRESTRequest` y `TRESTResponse` con la configuración necesaria para acceder al servicio que se tenga configurado en ese momento. Esa es la finalidad del botón **COPY COMPONENTS**. Tras hacer clic en él, cambiaríamos a un formulario Delphi y pegaríamos los componentes.

15.4 Acceso a servicios DataSnap con jQuery

Una aplicación multidispositivo, como la creada en el ejercicio de la sección previa, puede utilizarse desde distintos sistemas operativos y tipos de dispositivo para acceder al servidor DataSnap. Las plataformas objetivo serían todas aquellas para las que Delphi puede generar actualmente código. Dado que HTTP, REST y JSON son estándares, en realidad no estamos limitados a las soluciones que es posible desarrollar con Delphi, el servicio podría consumirse virtualmente desde cualquier lenguaje que permite efectuar peticiones HTTP y sea capaz de interpretar documentos JSON.

Entre las posibilidades existentes se encuentra el uso de JavaScript dentro de una página web. Si la combinamos con jQuery Mobile, la biblioteca pensada para el diseño de interfaces web móviles, el resultado será una aplicación accesible universalmente, desde cualquier navegador. Incluso existe la posibilidad, mediante herramientas como Apache Cordova, de empaquetar el código HTML/CSS/JS de forma que se instale y ejecute como una aplicación en iOS, Android, Windows Phone y otros sistemas operativos móviles.

En esta sección se muestra, mediante un sencillo ejercicio, cómo consumir desde una página jQuery Mobile un servicio REST desarrollado con Delphi.

15.4.1 Módulos proxy

Al enumerar los componentes agregados por el asistente al módulo web de un proyecto de servidor DataSnap REST (véase la Sección 15.1.2) se describió cómo el componente `TDSProxyGenerator` se encargaba de generar un módulo *proxy* que facilita el consumo del servicio desde otros lenguajes. En concreto, el agregado por defecto produce un *proxy* en lenguaje JavaScript. Este se almacena en el archivo `ServerFunctions.js`.

El *proxy* contiene una función por cada clase con métodos expuestos por el servidor DataSnap. El nombre de dicha función será el de la clase. Sus miembros serán funciones, con los nombres de los métodos Delphi, mediante las que se simplifica la llamada a dichos métodos, automatizando el procesamiento de los parámetros de entrada y de salida. En el *proxy* generado al ejecutar el servi-

dor DataSnap que ofrece el servicio de números aleatorios, que se almacenará en el archivo `ServerFunctions.js` alojado en la carpeta donde esté el ejecutable, encontraremos la siguiente definición:

```
function TRandomMethods(connectionInfo)
{
  this.executor = new ServerFunctionExecutor("TRandomMethods",connectionInfo);

  /*
   * @param FromN [in] - Type on server: Integer
   * @param ToN [in] - Type on server: Integer
   * @return result - Type on server: Integer
   */
  this.Rand = function(FromN, ToN) {
    var returnObject = this.executor.executeMethod('Rand', "GET", [FromN, ToN], arguments[2], true, arguments[3], arguments[4]);
    if (arguments[2] == null) {
      if (returnObject != null && returnObject.result != null && isArray(returnObject.result)) {
        var resultArray = returnObject.result;
        var resultObject = new Object();
        resultObject.FromN = FromN;
        resultObject.ToN = ToN;
        resultObject.result = resultArray[0];
        return resultObject;
      }
      return returnObject;
    }
  };

  this.Rand_URL = function(FromN, ToN) {
    return this.executor.getMethodURL("Rand", "GET", [FromN, ToN], arguments[2])[0];
  };

```

```
29    /*
30     * @param seed [in] - Type on server: Integer
31     * @return result - Type on server: Integer
32     */
33    this.updateRand = function(seed) {
34      ...
```

Listado 15.4 Parte del módulo *proxy* para JavaScript

Contando con este *proxy*, que enlazaríamos desde nuestra página web con la correspondiente etiqueta `script`, el esquema a seguir para invocar a los servicios usando JavaScript sería el mostrado a continuación:

```
1 var proxy = new TRandomMethods();
2 var output = proxy.Rand(1, 49);
3
4 alert(output.result);
```

Listado 15.5 Esquema de invocación de los métodos usando el *proxy*

En la documentación de Delphi se detalla cómo emplear los módulos *proxy* para ejecutar los métodos ofrecidos por el servidor. En los siguientes apartados de esta sección se muestra cómo hacer lo mismo sin necesidad de *proxy*, usando la biblioteca jQuery para realizar el trabajo.

15.4.2 Una interfaz jQuery Mobile

Generaremos la interfaz de usuario de nuestro cliente web usando la biblioteca jQuery Mobile. Esta incorpora una serie de etiquetas específicas que asocian cierta funcionalidad a los elementos a los que se asignan. Aunque está pensada específicamente para producir el mejor resultado posible en dispositivos con pantallas reducidas, como móviles y tabletas, la apariencia también será correcta en navegadores de escritorio.

El primer paso será crear un archivo, al que llamaremos `RESTClient.htm`, que estará estructurado en tres partes: cabecera de la página, elementos de la interfaz y código JavaScript. En la cabecera (elemento `head` del documento) enlazaremos la hoja de estilos de jQuery Mobile, la propia biblioteca jQuery

Mobile y también jQuery, sobre la que se asienta la anterior. Esta parte de la página sería la mostrada a continuación:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Cliente REST jQuery Mobile</title>
    <meta name="viewport"
      content="width=device-width, initial-scale=1">
    <link rel="stylesheet"
      href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css">
    <script src=
      "http://code.jquery.com/jquery-1.11.3.min.js">
    </script>
    <script src=
      "http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js">
    </script>
  </head>
  ...
```

Listado 15.6 Cabecera de la página web HTML5

Mediante la etiqueta `meta` con el nombre `viewport` ajustamos la escala de visualización de la interfaz, de forma que se utilice exactamente el ancho de la pantalla del dispositivo.

Para componer la interfaz recurriremos a un elemento de jQuery Mobile conocido como `collapsible`. Se trata de un elemento de interfaz que puede estar cerrado o abierto, mostrando en este último caso su contenido. Podemos incluir varios de estos elementos en un contenedor `collapsibleset` para tener varias zonas desplegables, de las cuales solo una estaría abierta en un momento dado.

La interfaz se dividirá en dos paneles de este tipo. El primero permitirá llamar a la función que facilita un número aleatorio, solicitando para ello los valores extremos del rango. Los controles de entrada serán los habituales en HTML5. Por

lo demás, la interfaz contará con un encabezado y un pie en los que nos limitaremos a mostrar un texto. La definición completa de la interfaz web sería la siguiente:

```
...
<body>
<div id="pageone" data-role="page">
  <div data-role="header">
    <h1>Rand REST Services</h1>
  </div>
  <div class="ui-content" data-role="main">
    <div data-role="collapsibleset">
      <div data-role="collapsible">
        <h3>Rand - GET</h3>
        <label for="MinValue">Límite inferior:</label>
        <input type="text" name="MinValue" id="MinValue">
        <label for="MaxValue">Límite superior:</label>
        <input type="text" name="MaxValue" id="MaxValue">
        <input type="button" value="Invocar" id="RandGET" ↙
            ↳ name="RandGET">
      </div>
      <div data-role="collapsible">
        <h3>Rand - POST</h3>
        <label for="SeedValue">Semilla:</label><input type="↙
            ↳ text" name="SeedValue" id="SeedValue">
        <input type="button" value="Invocar"
          id="RandPOST" name="RandPOST">
      </div>
    </div>
  </div>
  <div data-role="footer">
    <h1>Haga clic en el servicio a invocar</h1>
  </div>
</div>
...
```

Listado 15.7 Estructura de la interfaz de usuario

Si en este momento abrimos el documento en un navegador web deberíamos ver la interfaz de usuario, si bien esta no desencadena por el momento acción alguna al hacer clic sobre los botones de llamada a los servicios.

15.4.3 Invocaciones mediante AJAX

Para que al hacer clic sobre uno de los botones previos se llame al servicio implementado por el servidor DataSnap será preciso incluir en el documento el código correspondiente. Este puede residir en un módulo externo, que vincularíamos mediante una etiqueta `script` dispuesta al final del documento, o bien ser agregado directamente a este. Dado que en este caso dicho código no es muy extenso, optaremos por la segunda alternativa.

Al usar jQuery ciertas operaciones sobre el documento, como el acceso al contenido de los controles o la realización de llamadas de tipo AJAX, se ven considerablemente simplificadas respecto a cómo las haríamos en JavaScript, sin ninguna ayuda adicional. El siguiente listado corresponde a la función que se encargará de responder a la pulsación del primer botón existente en la interfaz de usuario:

```
<script>
$('#RandGET').click(function() {
    var min = $("#MinValue").val();
    var max = $("#MaxValue").val();

    $.ajax({
      type: "GET",
      url: "http://localhost:8080/datasnap/rest/↙
          ↳ TRandomMethods/Rand/" + min + "/" + max,
      success: function(data) {
        alert(JSON.parse(data).result);
      }
    });
});
...
```

Listado 15.8 Código JavaScript para obtener un número aleatorio

La sintaxis `$(#Identificador).miembro` se encarga de recuperar el elemento cuyo identificador se facilita y acceder al miembro indicado, ya sea para leer una propiedad o invocar a un método. En este caso la usamos para recuperar los valores introducidos en los dos recuadros de texto dispuestos en la interfaz de usuario.

Mediante la función `ajax()` de jQuery se simplifica la realización de llamadas de tipo AJAX, enviando solicitudes asíncronas cuyo destino, tipo y datos asociados se entregan a la función como una estructura de datos con múltiples campos. Esta estructura también puede incluir las funciones de respuesta a los eventos que se generarán una vez que el servidor responda. En el código previo se indica que la solicitud es de tipo `GET` y se entrega el URL del servidor, incluyendo los parámetros. La función asociada al evento `success` se limita a recuperar del documento JSON devuelto por el servidor el campo `result`, mostrándolo en un cuadro de diálogo emergente.

> **NOTA**
>
> *En `https://jquery.com/` puedes encontrar la documentación oficial de la API de jQuery, incluyendo multitud de ejemplos de uso. La documentación de jQuery Mobile está disponible en `https://jquerymobile.com/`.*

Para llamar al método que modifica la semilla del generador de números aleatorios tendremos que componer una solicitud algo más compleja, al tratarse de una solicitud de tipo `POST`. Los parámetros han de entregarse como si de un formulario se tratase, por esa razón se introducen en un objeto `FormData`. A la función `ajax()` habrá que facilitar también parámetros como `contentType` y `processData`, a fin de evitar que se incluya una cabecera de tipo de contenido y que manipule internamente la información. Por lo demás, el código será análogo al anterior: se extrae el valor introducido por el usuario en la interfaz, se efectúa la solicitud y se facilita una función con el código a ejecutar al recibir la respuesta.

En el siguiente listado se muestra la implementación de esta segunda función que, junto con la anterior, se introducirían al final del documento, antes de las etiquetas `</body>` y `</html>`, en un elemento `script`:

```
...
$('#RandPOST').click(function() {
    var seedValue = $("#SeedValue").val();
    var form = new FormData();
    form.append('seed', seedValue);

    $.ajax({
      type: "POST",
      url: "http://localhost:8080/datasnap/rest/↙
          ↳ TRandomMethods/Rand/" + seedValue,
      contentType: false,
      processData: false,
      data: form,
      success: function(data) {
        alert(JSON.parse(data).result);
      }
    });
});
</script>
```

Listado 15.9 Código JavaScript para cambiar la semilla

EJEMPLO 15.3 `RESTjQueryMobile`

Puedes encontrar la página `RESTClient.htm`, que incluye todo el código de los listados previos, en la carpeta `RESTjQueryMobile`.

15.4.4 Integración del cliente JavaScript en el servidor

Si abrimos la anterior aplicación, que es una página web, en nuestro navegador e intentamos acceder al servicio, asegurándonos de que el servidor esté en funcionamiento, comprobaremos que no se obtiene respuesta alguna. Podemos abrir

las herramientas para el desarrollador del navegador y comprobar qué ocurre: el navegador no permite que una página cargada localmente (el URL en la barra de direcciones es `file://...`) realice llamadas a un servidor[3], aunque este sea `localhost`.

Por otra parte, la interfaz jQuery Mobile que acabamos de diseñar ha de estar accesible para los clientes de alguna forma. Lo lógico es que resida en un servidor web, obteniéndose con una solicitud desde el navegador. Nuestro servidor DataSnap REST es capaz de servir este tipo de contenido, lo único que hemos de hacer es integrarlo. Para ello completaremos los pasos que se indican a continuación:

1. Abrimos el proyecto correspondiente al servidor DataSnap REST. Concretamente nos interesa abrir en el diseñador el módulo web, en el que están alojados los componentes introducidos por el asistente.

2. Agregamos al módulo web un componente `TPageProducer`, al que llamamos `RandServiceInvoker`. Introducimos en la propiedad `HTMLFile` la ruta y nombre del documento `RESTClient.htm` creado anteriormente (véase la Figura 15.16.)

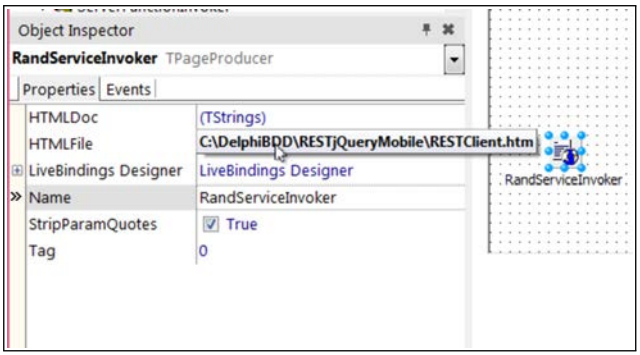


Figura 15.16 CONFIGURAMOS EL COMPONENTE `TPAGEPRODUCER`

[3]En general, el navegador no permitirá que el código embebido en una página efectúe llamadas a servidores que no sean el de origen, aquél del que proviene la propia página.

3. Para que el `TPageProducer` sea accesible a los cliente será preciso crear una nueva acción en el módulo web. Abrimos el editor asociado a la propiedad `Actions` del mismo, hacemos clic en el botón para añadir una nueva acción y establecemos las propiedades `Name`, `PathInfo` y `Producer` tal y como se ve en la Figura 15.17. El objetivo es que al usar la ruta `servidor/RandServiceInvoker` se desencadene la devolución de la página por parte del `TPageProducer`.

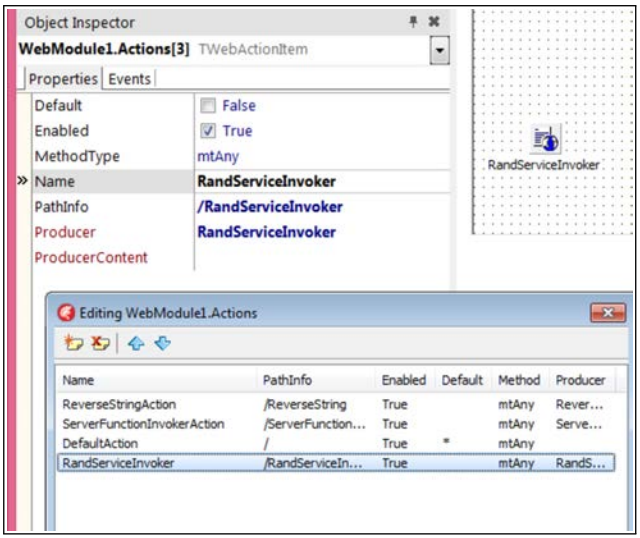


Figura 15.17 AGREGAMOS UNA NUEVA ACCIÓN AL MÓDULO WEB PARA DEVOLVER LA PÁGINA

4. Compilamos el proyecto del servidor y lo dejamos en ejecución en segundo plano.
5. Abrimos el navegador e introducimos el URL de solicitud de la página. En principio esta mostrará los dos paneles.
6. Hacemos clic sobre el panel de la función deseada, introducimos los parámetros de entrada y hacemos clic en **INVOCAR**. La respuesta del servidor se mostrará en una ventana flotante, como en la Figura 15.18.

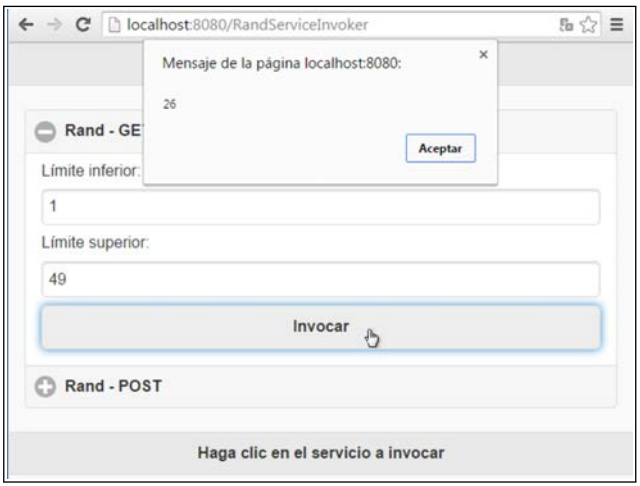


Figura 15.18 EL CLIENTE REST JQUERY MOBILE EN FUNCIONAMIENTO

15.5 DataSnap REST y conjuntos de datos

Al igual que un servidor DataSnap clásico, uno de tipo REST puede devolver conjuntos de datos, transfiriéndolos a los clientes para su visualización en la interfaz de usuario, y aceptar la lista de cambios para enviarlos al servidor de datos. La diferencia fundamental entre ambos tipos de servidores estriba en la representación utilizada para los datos. En los servidores DataSnap REST siempre se utiliza JSON.

Desde la versión XE5 Update2 Delphi cuenta con un mecanismo, denominado *FireDAC JSON Reflection*, que automatiza el proceso de conversión de conjuntos de datos FireDAC a JSON y viceversa. En esta última sección del capítulo vamos a desarrollar un ejercicio que se apoyará en dicho servicio, facilitando la edición remota de los datos desde los clientes. La estructura de ambos proyectos será prácticamente idéntica a la de los propuestos al final del capítulo previo.

15.5.1 FireDAC JSON Reflection

El núcleo de este servicio, disponible desde la versión XE5 Update 2 como se ha indicado, se encuentra en el módulo `Data.FireDACJSONReflect`. En él encontramos clases diseñadas para alojar conjuntos de datos y listas de conjuntos de datos en formato JSON, así como clases especializadas en la lectura y escritura de objetos derivados de `TDataSet`. Tendremos que agregar dicho módulo a la cláusula `uses` de los módulos de nuestro proyecto donde vayamos a usar algunas de las clases siguientes:

- `TFDJSONDataSets`: Un objeto de este tipo representa una lista de conjuntos de datos en formato JSON. Únicamente aporta un constructor, sin más propiedades ni métodos, actuando básicamente como un contenedor.
- `TFDJSONDataSetsWriter`: Es la clase encargada de escribir el contenido de un conjunto de datos en un objeto de clase `TFDJSONDataSets`. Habitualmente no crearemos objetos de esta clase, sino que usaremos el método de clase `ListAdd`. Este precisa dos parámetros: un objeto de clase `TFDJSONDataSets` y un derivado de `TFDDataSet`.
- `TFDJSONDataSetsReader`: Efectúa el trabajo complementario a la clase anterior, encargándose de leer conjuntos de datos alojados en un objeto `TFDJSONDataSets`. Disponemos de varios métodos de clase, entre ellos `GetListCount`, para obtener el número de elementos existentes en la lista; `GetListValue`, que facilita la recuperación de un elemento a partir de su índice, y `GetListValueByName` para leer un elemento por su nombre.
- `TFDJSONDeltas`: Al igual que `TFDJSONDataSets` esta clase actúa como entidad, representando el conjunto de campos pendientes de aplicar a un conjunto de datos. Puede escribirse en la entidad mediante los métodos de la clase `TFDJSONDeltasWriter` y leerse con los métodos de la clase `TFDJSONDeltasReader`.
- `TFDJSONDeltasApplyUpdates`: Mediante esta clase se transfiere la lista de cambios pendinetes alojada en un objeto `TFDJSONDeltas` a un conjunto de datos FireDAC.

Además de las clase que acaban de enumerarse, habitualmente también necesitaremos usar los componentes `TFDStanStorageXXXLink`, donde `X` puede

ser `Bin`, `JSON` o `XML`, dependiendo del formato en que se quiera almacenar la información. Bastará con agregar estos componentes al formulario, módulo de datos o módulo web donde precisemos utilizar los servicios FireDAC JSON Reflection.

15.5.2 Desarrollo del servidor

Veamos cómo, usando los componentes y clases mencionados en el anterior apartado, crearíamos un servidor DataSnap REST con capacidad para entregar un conjunto de datos y aceptar cambios enviados por los clientes.

El primer paso será usar el asistente que conocimos en una sección previa del capítulo, generando un nuevo proyecto con los elementos básicos. No es necesario que incluyamos los métodos de ejemplo ni los archivos web, únicamente la clase que contendrá los métodos del servidor. La clase que contendrá dichos métodos deberá ser una descendiente de `TDataModule`, no `TComponent`, lo cual nos permite disponer de un contenedor en el que incluir los componentes mostrados en la Figura 15.19

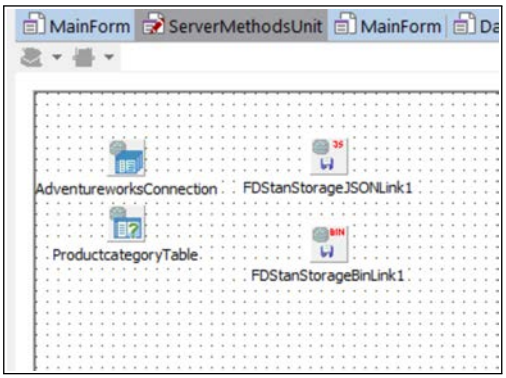


Figura 15.19 COMPONENTES A INCLUIR EN EL MÓDULO DE DATOS DEL SERVIDOR.

Tenemos un `TFDConnection` y un `TFDQuery` para acceder a los datos que deseamos poner a disposición de los clientes. Podemos usar el Explorador de datos para seleccionar la fuente, arrastrar una tabla hasta el módulo de datos y obtener estos dos componentes ya configurados. También incluiremos los com-

ponentes de almacenamiento binario y JSON mencionados previamente. Esto es todo lo que necesitamos.

El conjunto de datos no es directamente accesible para los clientes. Debemos agregar al servidor métodos que permitan obtenerlo y solicitar la aplicación de cambios. Introducimos estos métodos en la clase derivada de `TDataModule`, tal y como se muestra a continuación:

```
...
  public
    function GetProductCategories: TFDJSONDataSets;
    procedure ApplyChangesProductCategories(DeltaList: ↙
        ↳ TFDJSONDeltas);
...
```

Listado 15.10 Agregamos a la clase los dos métodos que expondrá el servidor

El método `GetProductCategories` devolverá como resultado un objeto `TFDJSONDataSets`, con la lista de conjuntos de datos desde la que se extraerá el que nos interesa. En cuanto al segundo método, precisa como argumento un objeto `TFDJSONDeltas` con la lista de cambios a aplicar.

Para facilitar al cliente los datos obtenidos por el `TFDQuery` hemos de escribir el contenido de este en un `TFDJSONDataSets`, transformándolo en un documento JSON. Para ello lo entregaremos como argumento al método `ListAdd` de la clase `TFDJSONDataSetsWriter`, tal y como se muestra a continuación. El objeto `TFDJSONDataSets` sería el resultado a devolver, conteniendo los datos de la tabla elegida:

```
...
function TProductInfo.GetProductCategories: TFDJSONDataSets;
var
  dataset: TFDJSONDataSets;
begin
  ProductcategoryTable.Open;
  dataset := TFDJSONDataSets.Create;
  TFDJSONDataSetsWriter.ListAdd(dataset, ↙
      ↳ ProductcategoryTable);
```

```
9
10    result := dataset;
11  end;
```

Listado 15.11 Implementación del método de obtención del conjunto de datos

En cuanto al método de aplicación de cambios, ha de convertir el JSON recibido del cliente, un objeto `TFDJSONDeltas`, en un conjunto de cambios que pueda ser aplicado al `TFDQuery`. Con este fin recurriremos a la clase `TFDJSONDeltasApplyUpdates`, facilitando la lista de cambios como parámetro e invocando después a su método `ApplyUpdates`. La implementación sería la siguiente:

```
1  ...
2  procedure TProductInfo.ApplyChangesProductCategories(↙
       ↳ DeltaList: TFDJSONDeltas);
3  var
4    listChanges: IFDJSONDeltasApplyUpdates;
5  begin
6    listChanges:= TFDJSONDeltasApplyUpdates.Create(DeltaList);
7    listChanges.ApplyUpdates('ProductcategoryTable', ↙
         ↳ ProductcategoryTable.Command);
8  end;
```

Listado 15.12 Implementación del método de aplicación de cambios

EJEMPLO 15.4 `RESTCategories.dproj`

Puedes encontrar este proyecto completo en la carpeta `RESTCategories`. El proyecto está configurado para conectar con la base de datos SQL Server `AdventureWorks`. En caso necesario podemos modificar la conexión y consulta para utilizar otra base de datos.

Ahora ya podemos compilar, ejecutar y poner en marcha el servidor. Probar el funcionamiento del primero de sus métodos, el que devuelve el conjunto de datos, resultará fácil. No tenemos más que abrir el navegador e introducir el URL correspondiente, como se ha hecho en la Figura 15.20. Como puede apreciarse, el documento JSON tiene una cabecera con los datos descriptivos de los conjuntos de datos que contiene. La información de estos se ha codificado para evitar problemas al transferirla a través de la red.

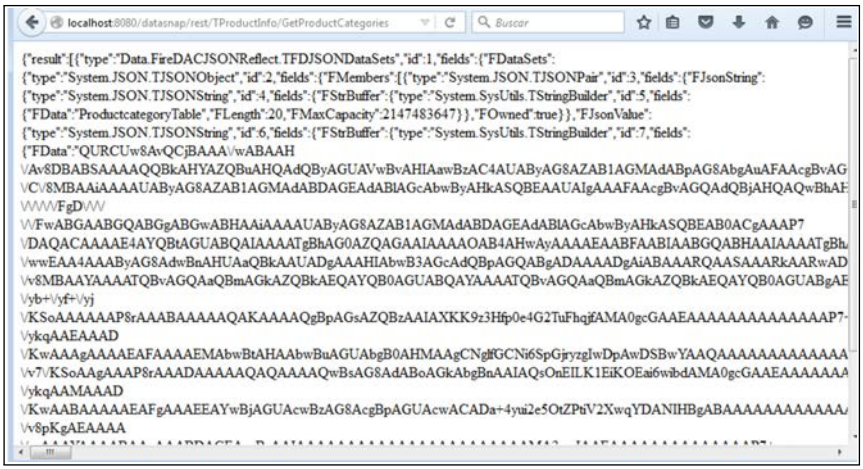


Figura 15.20 DOCUMENTO JSON CONTENIENDO EL CONJUNTO DE DATOS

Componer una solicitud para invocar al segundo método sería bastante más complejo, aunque podríamos hacerlo utilizando el depurador REST siempre que fuésemos capaces de escribir el documento JSON a enviar al servidor.

15.5.3 Desarrollo del cliente

Al crear un cliente REST que precisa trabajar con conjuntos de datos FireDAC recurriremos a los componentes y clases que ya hemos usado en el servidor, pero básicamente efectuando las operaciones a la inversa. Asumiendo que hemos generado un nuevo proyecto de tipo multidispositivo, los pasos a seguir serían:

1. Abrimos el cuadro de diálogo **NEW ITEMS** y elegimos de la carpeta **DATA SNAP SERVER** el elemento `DataSnap REST Client Module` (véase la Figura 15.21). Esto añadirá al actual proyecto dos módulos. En el módulo `ClientClassesUnit` encontraremos la implementación de una clase derivada de `TDSAdminRestClient`, con los métodos necesarios para invocar a los métodos del servidor. El segundo método ofrece una vía de acceso rápido a un objeto de la clase anterior.

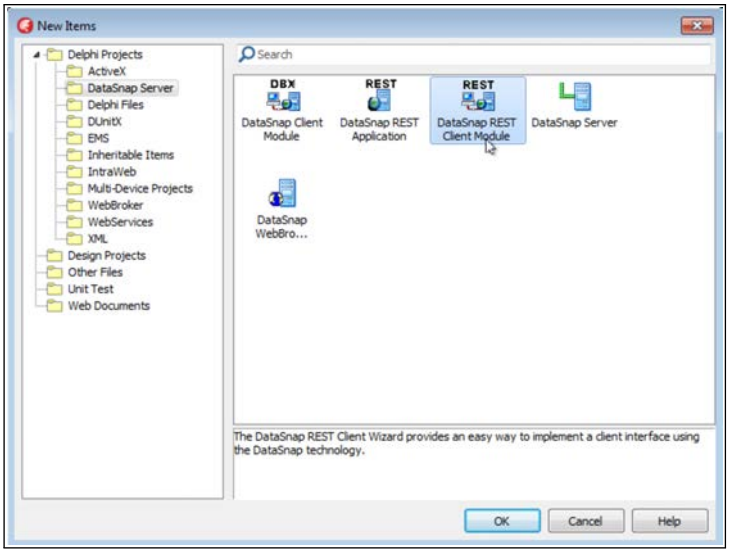


Figura 15.21 EL PROYECTO NECESITA UN MÓDULO REST DE CLIENTE PARA ACCEDER AL SERVIDOR

2. Agregamos al proyecto un módulo de datos. Este será el encargado de alojar los componentes de acceso a los datos y el código asociado. Mediante la opción **FILE—USE UNIT** añadimos una referencia a los módulos cliente creados en el paso anterior, de forma que podamos acceder a su contenido desde el módulo de datos.

3. Incluimos en el módulo de datos un `TFDMemTable` a cuya propiedad `CachedUpdates` daremos el valor `True`. De esta forma los cambios

se almacenarán localmente. También añadiremos al módulo un componente `TFDStanStorageJSONLink` y un `TFDStanStorageBinLink`, agregando así las referencias necesarias.

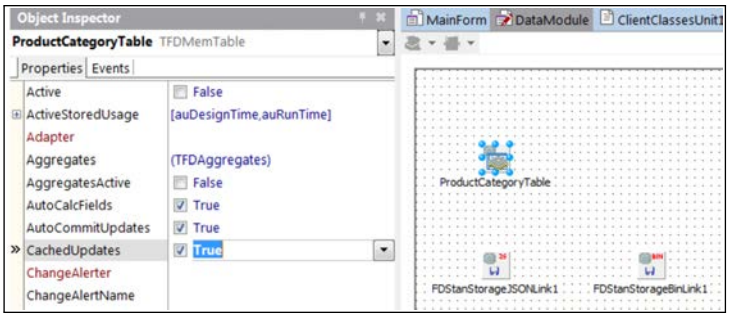


Figura 15.22 COMPONENTES EN EL MÓDULO DE DATOS

4. El módulo de datos contendrá dos métodos públicos. El primero se encargará de obtener del servidor REST el conjunto de datos remoto, incluyendo su contenido en el `TFDMemTable` local. El segundo enviaría los cambios registrados por este de vuelta al servidor. La implementación de estos dos métodos es la siguiente:

```
1  ...
2  procedure TDataModule2.GetProductCategories;
3  var
4    listDatasets: TFDJSONDataSets;
5    datasetProduct: TFDDataSet;
6  begin
7    listDatasets := ClientModule1.ProductInfoClient.↙
         ↳ GetProductCategories;
8    datasetProduct := TFDJSONDataSetsReader.↙
         ↳ GetListValueByName(
9      listDatasets, 'ProductcategoryTable');
10
11   ProductCategoryTable.Close;
12   ProductCategoryTable.AppendData(datasetProduct);
```

```
13 end;
14
15 procedure TDataModule2.ApplyChangesProduct;
16 var
17   listChanges: TFDJSONDeltas;
18 begin
19   listChanges := TFDJSONDeltas.Create;
20   TFDJSONDeltasWriter.ListAdd(listChanges,
21     'ProductcategoryTable',
22     ProductDataModule.ProductCategoryTable);
23   ClientModule1.ProductInfoClient.↙
       ↳ ApplyChangesProductCategories(listChanges);
24 end;
```

Listado 15.13 Métodos del módulo de datos

5. El siguiente paso será componer la interfaz de usuario en el formulario con que cuenta el proyecto. Esta constará de un control `TGrid`, que mostrará los datos y facilitará su edición; un `TPanel` y un `TButton` para aplicar los cambios pendientes.

6. Agregamos al formulario una referencia al módulo de datos, a fin de poder vincular el `TFDMemTable` con el `TGrid`, por una parte, y poder llamar a los métodos anteriores, por otra.

7. Usamos el diseñador visual **LIVEBINDINGS** para conectar el `TFDMemTable` con la cuadrícula de datos.

8. Seleccionamos el formulario, localizamos su evento `OnShow`, hacemos doble clic sobre él e introducimos el código siguiente. De esta forma se recuperará del servidor el conjunto de datos y la cuadrícula mostrará su contenido:

```
1 ...
2 procedure TfrmMain.FormShow(Sender: TObject);
3 begin
4   ProductDataModule.GetProductCategories;
5 end;
```

Listado 15.14 Al mostrar el formulario se obtiene el conjunto de datos

9. Finalmente haremos doble clic sobre el botón, abriendo el método que se encargará de solicitar al módulo de datos la transferencia de los cambios al servidor:

```
1 ...
2 procedure TfrmMain.btnAplicarCambiosClick(Sender: TObject↙
      ↳ );
3 begin
4   ProductDataModule.ApplyChangesProduct;
5 end;
```

Listado 15.15 El botón transferirá los cambios pendientes

Con esto tenemos nuestro cliente REST terminado. Podemos ejecutarlo y, siempre que el servidor esté en funcionamiento, para ver en la cuadrícula el conjunto de datos seleccionado en el servidor. Al introducir un cambio y hacer clic en el botón (véase la Figura 15.23) este se enviará al servidor que, a su vez, lo entregará al RDBMS.

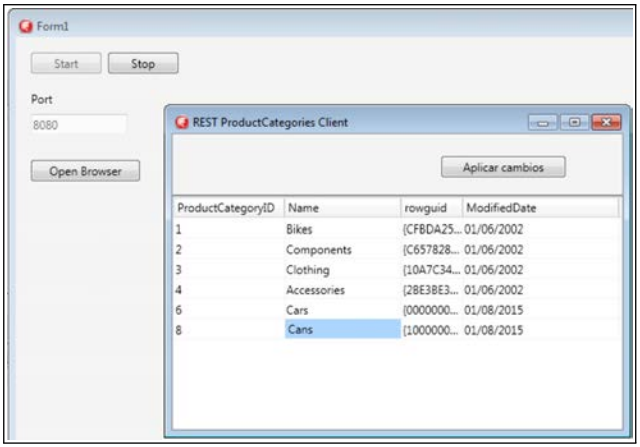


Figura 15.23 EL CLIENTE PUEDE MODIFICAR LOS DATOS FACILITADOS POR EL SERVIDOR REST

15.6 Resumen

En este capítulo hemos conocido los asistentes, componentes y clases que incorpora Delphi con el objetivo de desarrollar servidores y clientes REST. Un servidor DataSnap REST expone servicios que son accesibles mediante el protocolo estándar HTTP, según los principios REST y usando JSON como formato para el transporte de los datos. De esta forma se garantiza el acceso desde prácticamente cualquier sistema, lenguaje de programación y herramienta.

Para terminar esta parte del libro, en el capítulo siguiente se facilita una introducción a EMS, una de las novedades más interesantes incluidas en las últimas versiones de Delphi.

Capítulo 16

INTRODUCCIÓN A EMS

En el capítulo previo hemos comprobado en la práctica, con un par de ejercicios sencillos, cómo podemos crear servicios accesibles mediante peticiones REST. Los componentes de DataSnap, sus asistentes y los componentes asociados a JSON para la conversión y transferencia de conjuntos de datos se ocupan de una buena parte del trabajo.

Un servidor de aplicaciones real precisará otras funciones, como son la autenticación de los usuarios, la autorización de acceso a los servicios y herramientas de administración que permiten al responsable del servidor gestionar las cuentas de usuario y los grupos de funcionalidad a las que pertenecen. Para algunas de

estas tareas DataSnap cuenta con componentes específicos, pero otras requerirán un trabajo adicional considerable por parte del desarrollador.

Este capítulo introduce una de las novedades incluidas en las últimas versiones de Delphi, concretamente a partir de la versión XE7: EMS (*Enterprise Mobility Services*). La empresa lo define como un *middleware*, una capa de software cuyo objetivo es hacer más fácil el desarrollo de nuevos servidores de aplicaciones. A continuación se describe qué funcionalidad aporta EMS, cómo acceder a sus servicios fundamentales y cómo desarrollar paquetes EMS para extender dicha funcionalidad.

16.1 Características de EMS

A diferencia de DataSnap, que es una tecnología que no conlleva coste adicional y se incorpora a nuestros proyectos a través de los componentes que ya conocemos, EMS es un servidor independiente, un software por el que es preciso pagar una licencia para poder usarlo en explotación. Delphi incluye una versión de desarrollo de dicho servidor, capaz de ejecutarse de forma autónoma y que facilita la construcción y prueba de nuevas soluciones. El servidor para explotación es un módulo IIS, por lo que únicamente puede utilizarse en un equipo con Windows y el servidor web de Microsoft.

Los servicios ofrecidos por EMS son accesibles por HTTP[1], con peticiones REST, y se utiliza JSON para la representación de los datos. Las técnicas que se explicaron en el capítulo previo para usar servicios REST, desde páginas web, con los componentes REST para clientes Delphi, etc., podríamos también usarlas para trabajar con EMS.

16.1.1 Funcionalidad de EMS

EMS cuenta con los siguientes servicios y herramientas fundamentales. Cada servicio cuenta con un grupo de lo que en la documentación de EMS se denomina *endpoint*, es decir, métodos individuales a los que es posible invocar mediante llamadas de tipo REST:

[1] Si se activa el cifrado en la comunicación el protocolo sería HTTPS en lugar de HTTP.

- **Administración de usuarios**: Es un servicio REST que permite enumerar los usuarios, obtener detalles de ellos, agregar y eliminar usuarios. Cada operación está asociada a un *endpoint*.
- **Administración de grupos**: Se trata de un servicio REST análogo al anterior, facilitando la gestión de grupos de usuarios.
- **Servicios de seguridad**: EMS cuenta con los servicios necesarios para que los usuarios se identifiquen antes de poder acceder a las funciones ofrecidas por el servidor (autenticación), controlando asimismo los permisos que tiene cada usuario, según sus privilegios y los de los grupos a los que pertenezca, para acceder a los servicios (autorización). También existe la posibilidad de cifrar toda la comunicación entre clientes y servidor EMS mediante HTTPS.
- **Datos estadísticos**: El servidor va recopilando información de uso mientras está en funcionamiento, permitiendo así la recuperación de datos estadísticos diversos.
- **Consola de administración**: Es una herramienta que permite al desarrollador examinar la base de datos de EMS y las estadísticas recopiladas por el servidor.

16.1.2 Herramientas de desarrollo

Para probar los servicios que ofrece EMS, así como para desarrollar paquetes que extiendan su funcionalidad, no precisamos recurrir al servidor completo, el que sería desplegado en el entorno de explotación. En la carpeta de binarios donde tengamos instalado Delphi (véase la Figura 16.1) encontraremos las siguientes herramientas de desarrollo:

- `EMSDevServer`: Este ejecutable es la versión de desarrollo del servidor EMS. Lo utilizaremos para realizar pruebas de los servicios que ofrece y desarrollar nuevos paquetes EMS. En la siguiente sección se explica cómo configurar y usar este servidor de desarrollo.
- `EMSDevConsole`: Este programa actúa como un servidor web que ofrece una consola básica de supervisión del servidor EMS. Para acceder a ella se emplea una cuenta de usuario específica, permitiendo examinar los grupos, usuarios, número de instalaciones e información estadística del servidor.

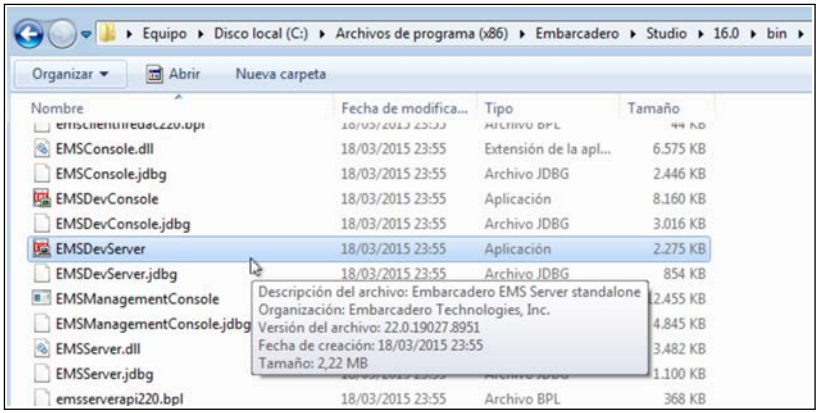


Figura 16.1 EJECUTABLES CON LA VERSIÓN DE DESARROLLO DEL SERVIDOR Y CONSOLA EMS

El funcionamiento del servidor viene determinado por un conjunto de parámetros de configuración. Estos se alojan en un archivo, cuya localización está anotada en el registro de Windows, con el nombre `emsserver.ini`.

> **NOTA**
>
> *EMS también incluye una versión de desarrollo de InterBase que será la utilizada por defecto para almacenar los datos de configuración del servidor, incluyendo los grupos y usuarios.*

16.1.3 Componentes para aplicaciones

Además de las herramientas anteriores, EMS también aporta a Delphi un conjunto de componentes (los encontraremos en la página **BAAS CLIENT** de la Paleta de herramientas) específico que facilita el acceso a los servicios que ofrece el servidor (véase la Figura 16.2). Con estos pueden gestionarse los usuarios y grupos, efectuar el proceso de autenticación o simplificar la transferencia de conjuntos de datos entre servidor EMS y clientes, entre otras tareas.

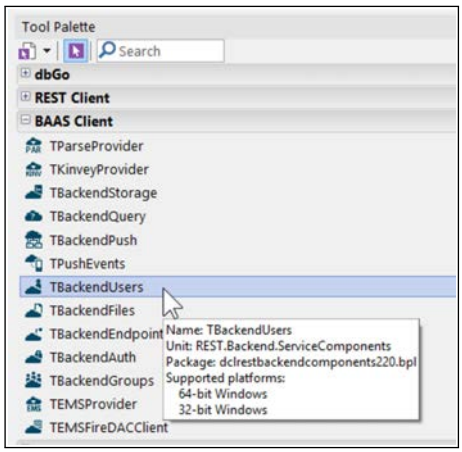


Figura 16.2 COMPONENTES PARA ACCEDER A LOS SERVICIOS DE EMS

El entorno de Delphi también incluye asistentes específicos, concretamente los dos mostrados en la Figura 16.3, cuyo objetivo es facilitar el desarrollo de paquetes EMS. Con estos se extiende la funcionalidad del servidor EMS.

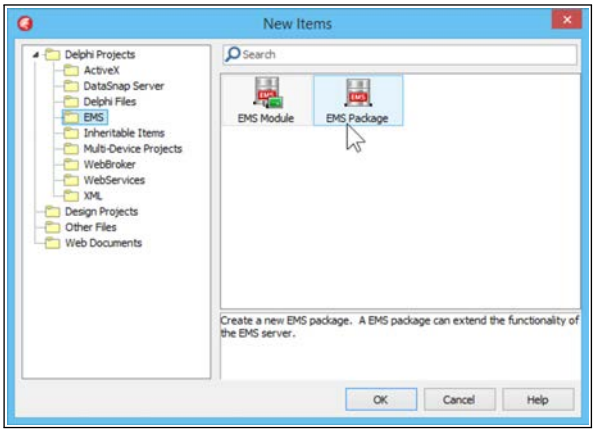


Figura 16.3 ASISTENTES PARA DESARROLLO DE PAQUETES EMS

16.2 El servidor EMS

En este punto tenemos una visión general abstracta de qué es y qué ofrece el servidor EMS. Veamos ahora en la práctica cómo poner en marcha el servidor de desarrollo EMS, cómo configurarlo y cómo usar la consola de desarrollo y algunos de los servicios propios de EMS.

16.2.1 Configuración del servidor EMS

Para poner en marcha el servidor de desarrollo EMS localizaremos el archivo `EMSDevServer.exe`, indicado antes, y lo ejecutaremos. Al ser la primera ejecución aparecerá un cuadro de diálogo indicando que no se encuentra el archivo de configuración, aún no existe, e invitándonos a iniciar el asistente de configuración, a lo que responderemos afirmativamente.

El mencionado asistente consta de cuatro pasos, cuya finalidad es:

1. En el primer paso, al que corresponde la Figura 16.4, se establecen los parámetros de creación de la base de datos que alojará la información de EMS: grupos, usuarios, etc. Si tenemos instalada la edición de desarrollo de InterBase incluida con Delphi, que es lo habitual, podemos aceptar la configuración propuesta por defecto y hacer clic en **NEXT**.
2. Inicialmente la lista de grupos y usuarios estará vacía salvo que marquemos las opciones ofrecidas por el asistente en el segundo paso. Esto creará un grupo y un usuario que nos permitirán realizar pruebas en el entorno de desarrollo.
3. La consola de desarrollo de EMS cuenta con su propias credenciales de acceso. Estas se establecen en el tercer paso del asistente. Es recomendable que cambiemos las propuestas por defecto.
4. El último paso del asistente es de confirmación, indicando la ruta y nombre de la base de datos, el archivo de configuración y entrada en el registro de Windows que se crearán al hacer clic en el botón **FINISH**.

Si el proceso concluye satisfactoriamente, y no se produce ningún error al configurar la base de datos o crear el archivo de configuración, el asistente finalizará con la confirmación mostrada en la Figura 16.5. Ahora el servidor EMS

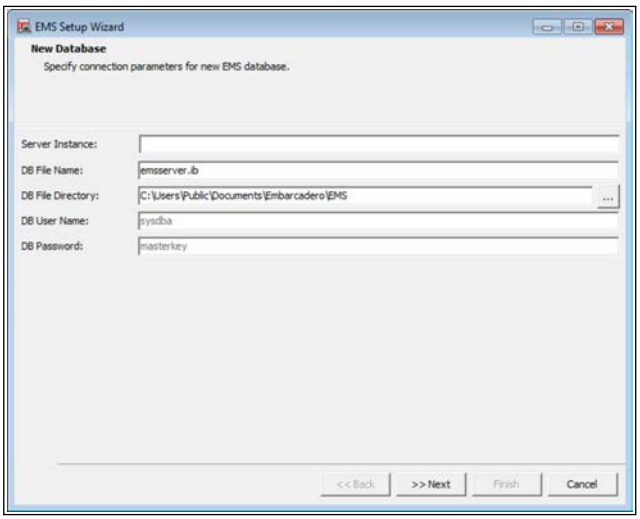


Figura 16.4 PRIMER PASO DEL ASISTENTE DE CONFIGURACIÓN DE EMS

está preparado para comenzar a funcionar. En cualquier momento podemos acceder al archivo `emsserver.ini` para modificar la configuración establecida, por ejemplo para activar la comunicación por HTTPS o modificar los parámetros de seguridad de acceso a los recursos.

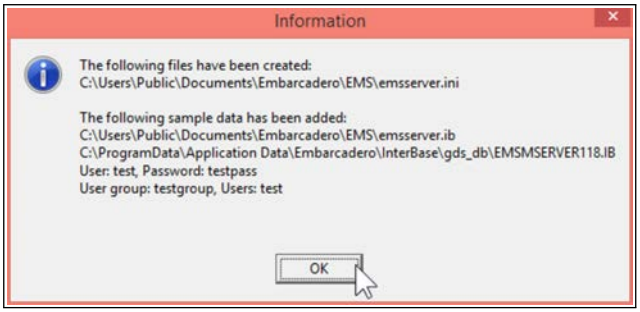


Figura 16.5 EL ASISTENTE CONFIRMA LA CONFIGURACIÓN DE EMS

16.2.2 Consola de control del servidor

Finalizada la configuración, cada vez que ejecutemos el servidor EMS de desarrollo veremos aparecer una interfaz de usuario como la de la Figura 16.6. Esta es la consola de control del servidor. Los botones dispuestos en la parte superior nos permiten iniciar y detener el servidor, así como abrir una ventana de navegador y lanzar la consola de desarrollo. También se permite cambiar el puerto de comunicación por defecto.

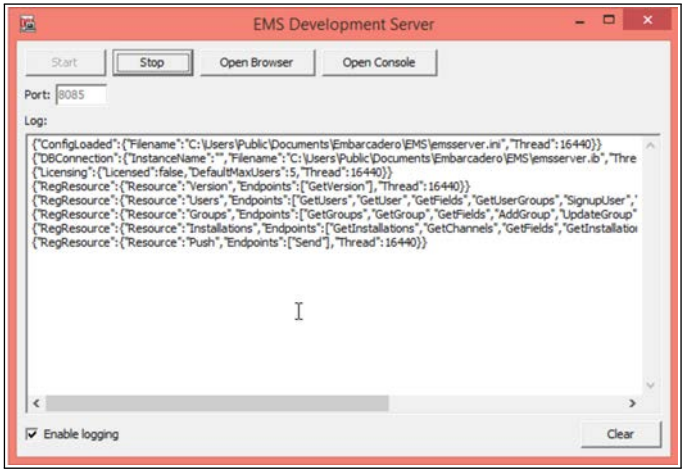


Figura 16.6 CONSOLA DE CONTROL DEL SERVIDOR EMS

La mayor parte de la interfaz de la consola está ocupada por una lista en la que aparece el registro de eventos del servidor EMS. En principio solo veremos el registro de los diferentes módulos de servicio de EMS. A medida que lo utilicemos también irán apareciendo entradas relativas a los servicios a los que se ha invocado.

16.2.3 La consola de desarrollo

Hagamos clic en el botón **OPEN CONSOLE** de la consola de control, o alternativamente podemos ejecutar directamente el programa `EMSDevConsole.exe`,

para abrir la consola de desarrollo. La interfaz de control de esta consola es muy simple, como puede apreciarse en la Figura 16.7. En general, nos limitaremos a iniciar la consola, haciendo clic en el botón **START**, y a abrirla a continuación en el navegador web. De esta forma accederemos a la verdadera consola de desarrollo.

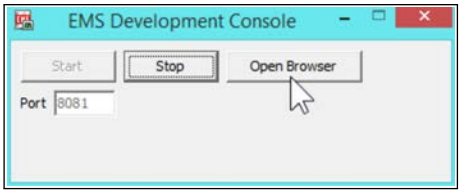


Figura 16.7 PROGRAMA DE CONTROL DE LA CONSOLA DE DESARROLLO

Tras iniciar sesión, usando para ello las credenciales específicas de la consola que establecimos en el asistente, podremos examinar los usuarios y grupos, así como acceder a los datos analíticos de uso del servidor (véase la Figura 16.8).

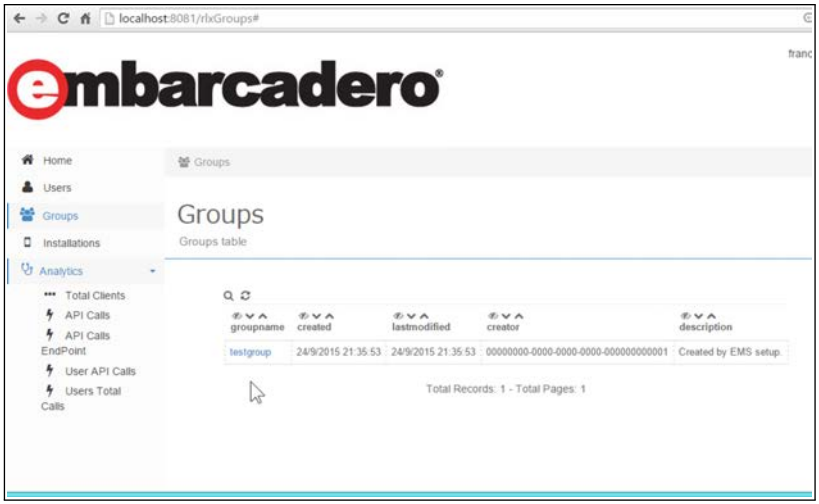


Figura 16.8 CONSOLA DE DESARROLLO DE EMS

Esta consola facilita, sobre todo, datos estadísticos de uso de las distintas API del servidor, analizados por fechas y horas, lo cual nos permitirá supervisar la actividad general del servidor EMS una vez que lo tengamos configurado en un entorno de explotación.

16.3 Servicios ofrecidos por EMS

Asumiendo que tenemos en funcionamiento el servidor EMS de desarrollo[2], en esta sección se enumeran los servicios que ofrece por defecto y describe con ejemplos el uso de algunos de ellos.

Si examinamos el registro de eventos que aparece en la consola de control del servidor, cuyo detalle se muestra en la Figura 16.9, encontraremos varias entradas de tipo `RegResource`. Son los eventos de registro de recursos en el servidor EMS, en los que se detalla el nombre del recursos y los *endpoints* o funciones a los que podemos invocar.

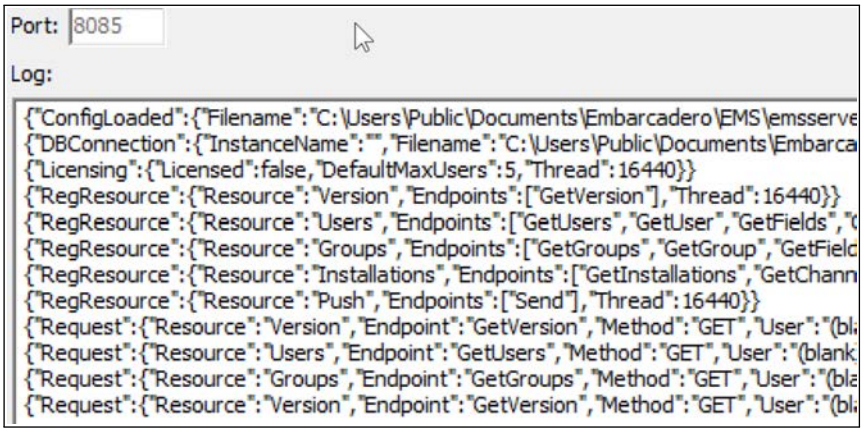


Figura 16.9 EN EL REGISTRO DE EVENTOS APARECEN LOS RECURSOS REGISTRADOS

[2]Lo explicado aquí para este servidor de desarrollo sería igualmente aplicable al servidor final puesto en explotación.

En los siguiente apartados se enumeran los métodos con que cuenta cada uno de los recursos esenciales de EMS.

16.3.1 Gestión de grupos

El recurso EMS para gestión de grupos de usuarios se denomina `Groups`, por lo que accederemos a él utilizando un URL con la estructura `http://servidor-EMS/Groups/parámetros`, como es habitual en un servicio REST. Se contempla el uso de peticiones `GET`, `POST`, `PUT` y `DELETE` para obtener la lista de grupos o información de un grupo, agregar un nuevo grupo, modificar datos de un grupo existente o eliminar un grupo, respectivamente. El nombre del grupo del que se quieren obtener detalles, a modificar o a eliminar se facilitará como parámetro. El resto de información se enviará y devolverá en el cuerpo de la solicitud, como documento JSON.

Para invocar a las funciones accesibles mediante peticiones `GET` podemos usar directamente un navegador. Asumiendo que el servidor EMS está en funcionamiento, una solicitud a `http://servidor-EMS/Groups` devolvería la lista de grupos registrados en el servidor (véase la Figura 16.10).

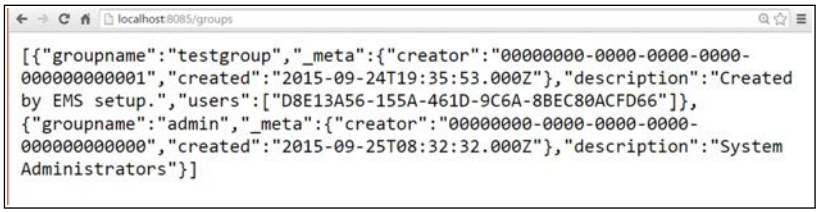


Figura 16.10 RECUPERAMOS LA LISTA DE GRUPOS REGISTRADOS

Conociendo el nombre de un grupo, podemos solicitar al servidor los detalles del mismo simplemente agregando dicho nombre como parámetro al URL anterior, tal y como se muestra en la Figura 16.11.

Para invocar a las demás funciones tendremos que recurrir a una herramienta que nos permita enviar solicitudes de otros tipos, ya que con el navegador, sin preparar una página específica, solo podemos enviar peticiones `GET`. Dicha herramienta puede ser el depurador REST integrado en Delphi que usábamos en el capítulo previo.

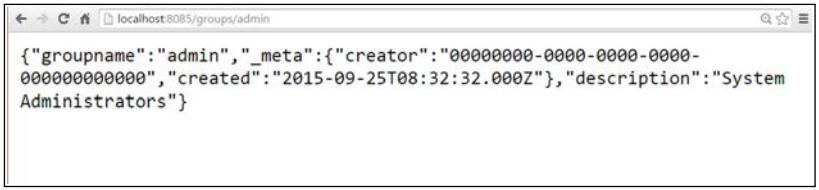


Figura 16.11 OBTENEMOS DETALLES DE UN GRUPO CONCRETO

Supongamos que queremos crear un nuevo grupo, inicialmente solo existe `testgroup`, al que llamaremos `admin`. Abrimos el depurador REST, introducimos el URL, seleccionamos el método `POST` para enviar la petición y facilitamos el JSON con el nombre y descripción del grupo, como se hace en la Figura 16.12.

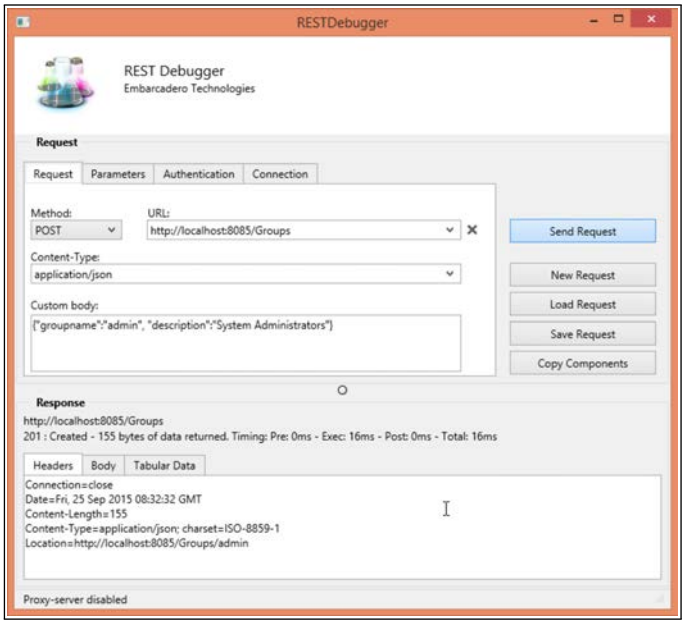


Figura 16.12 CREAMOS UN NUEVO GRUPO CON LA SOLICITUD DE TIPO `POST`

Tras enviar esta solicitud podemos solicitar de nuevo la lista de grupos, o bien usar la consola de desarrollo indicada antes, a fin de comprobar el resultado. De forma análoga usaríamos los servicios de actualización y eliminación, configurando el tipo de petición, agregando al URL el nombre del grupo y facilitando, en el caso de la actualización, el JSON con los nuevos valores.

16.3.2 Gestión de usuarios

Mediante el recurso de gestión de usuarios de EMS, llamado `Users`, podemos enumerar los usuarios, recuperar detalles de un usuario, agregar usuarios, modificarlos y eliminarlos. Los fundamentos son exactamente los mismos descritos antes para los grupos.

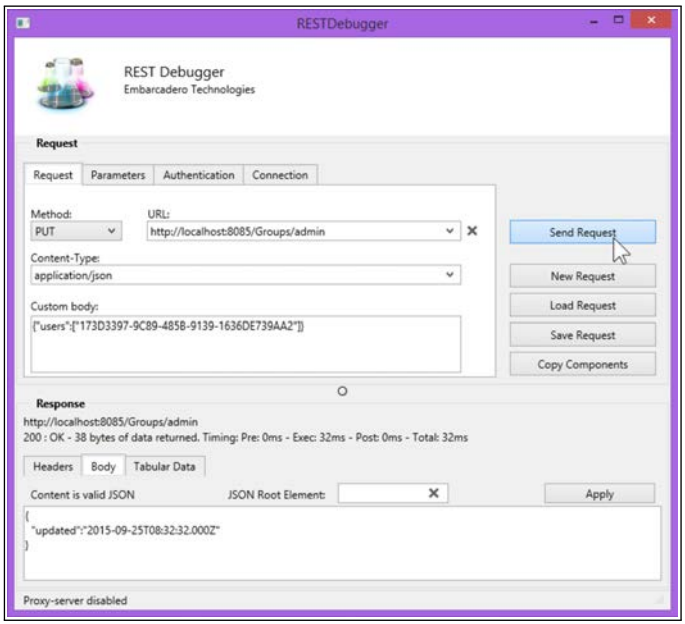


Figura 16.13 AÑADIMOS UN USUARIO NUEVO AL GRUPO CREADO ANTES

Al registrar un nuevo usuario se le asignará automáticamente un identificador único, generado por el servidor EMS. Podemos usarlo para vincular el usuario

con los grupos a los que deseamos que pertenezca. Con este fin usaríamos una solicitud de tipo `PUT` modificando la lista de usuarios del grupo, tal y como se muestra en la Figura 16.13.

Este recurso de EMS también dispone de servicios de autenticación de usuarios, lo cual permite acceder a servicios que estén restringidos a un cierto usuario o grupo de usuarios. El procedimiento a seguir consta de los siguientes pasos:

1. Usando el sufijo `/USers/Login` procedemos a iniciar sesión. Para ello se envía una solicitud `POST` con un documento JSON que incluya dos datos: `username`, con el nombre del usuario, y `password`, con la contraseña, tal y como se aprecia en la parte superior de la Figura 16.14.

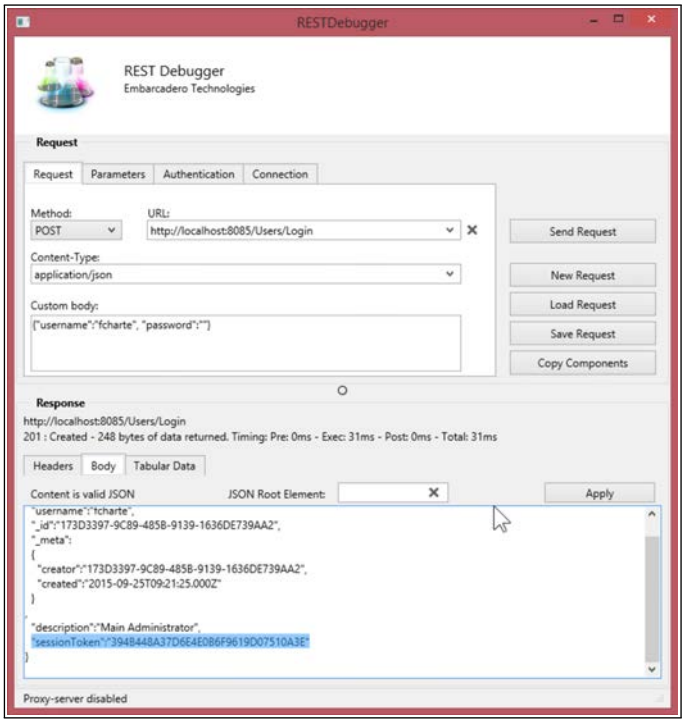


Figura 16.14 INICIAMOS SESIÓN EN EL SERVIDOR EMS

2. Si el usuario existe y la contraseña es correcta se completará el inicio de sesión. El servidor responderá devolviendo un documento JSON del que nos interesa específicamente el campo `sessionToken` (véase la parte inferior de la Figura 16.14, resaltado del resto). Se trata de una secuencia alfanumérica generada a partir del inicio de sesión y con validez temporal que deberemos conservar durante la sesión.

3. En las siguientes solicitudes que efectuemos deberemos incluir en la cabecera HTTP un campo con el nombre `X-Embarcadero-Session-Token`, asignándole el *token* obtenido en el paso previo. Esto permitirá al servidor EMS saber quiénes somos, ya que nadie más puede tener esa información. En el depurador REST usaríamos la página **PARAMETERS**, como puede verse en la Figura 16.15.

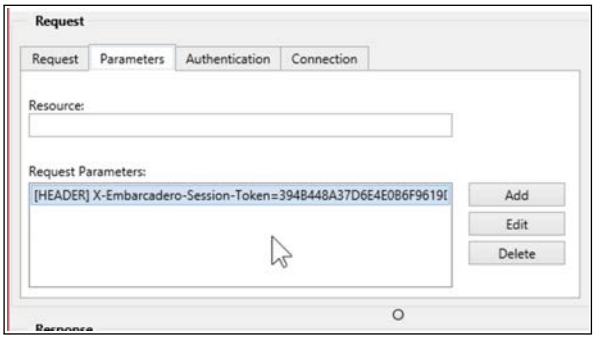


Figura 16.15 FACILITAMOS AL SERVICIO EL *token* OBTENIDO TRAS INICIAR SESIÓN

Podemos examinar el registro de eventos de la consola de control de EMS (véase la Figura 16.16) para comprobar la diferencia entre las llamadas sin autenticación, hechas anteriormente, y aquellas que incluyen el *token* obtenido tras el inicio de sesión.

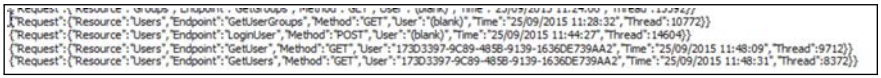


Figura 16.16 EN EL REGISTRO SE APRECIA EL INICIO DE SESIÓN

16.3.3 Otros servicios

Además de los ya mencionados, el servidor EMS cuenta también con un recurso que nos permite obtener información sobre las aplicaciones que se han registrado para obtener notificaciones *push* por parte del servidor. Por cada registro se devolverán datos como el tipo de dispositivo y su identificador o la lista de canales a los que está suscrito. Este recurso se identifica como `Installations` y contempla fundamentalmente las mismas operaciones enumeradas antes para grupos y usuarios.

Figura 16.17 CAMPOS ASOCIADOS AL REGISTRO DE INSTALACIONES

Otro de los servicios básicos es `Version`, encargado de devolver la versión del servidor EMS con el que está trabajando la aplicación.

16.4 Componentes Delphi para operar con EMS

Los servicios de cualquier servidor REST, incluyendo los que ofrece EMS, pueden ser consumidos desde Delphi mediante los componentes de la biblioteca REST para aplicaciones cliente que conocimos en el capítulo previo. En la Paleta de herramientas, no obstante, encontramos un conjunto de componentes específicos para operar sobre EMS. La mayor parte de ellos se encuentran en la página **BAAS CLIENT** (véase la Figura 16.2, en página 473).

De estos, los componentes esenciales y la finalidad de cada cada uno de ellos son:

- `TEMSProvider`: Es el componente encargado de facilitar a los demás la conexión con el servidor EMS, centralizando los datos relativos a su locali-

zación, protocolo de comunicación, etc. Como mínimo será preciso facilitar valores para las propiedades `URLHost` y `URLPort`, especificando el nombre de la máquina y puerto en que se ejecuta el servidor EMS.

- `TBackendGroups` y `TBackendUsers`: Se conectan al anterior mediante la propiedad `Provider`. El primero cuenta con una propiedad `Groups` y el segundo con una propiedad `Users`, cada una de las cuales da acceso a una interfaz con métodos que permiten obtener los datos de un usuario o grupo, agregar nuevas entradas, modificar las existentes, vincular usuarios a grupos, etc. Los métodos son en su mayor parte descriptivos, como se aprecia en la Figura 16.18. No tenemos más que facilitar los parámetros adecuados y procesar la salida, sin preocuparnos por conversiones a y desde JSON o detalles de las llamadas REST.

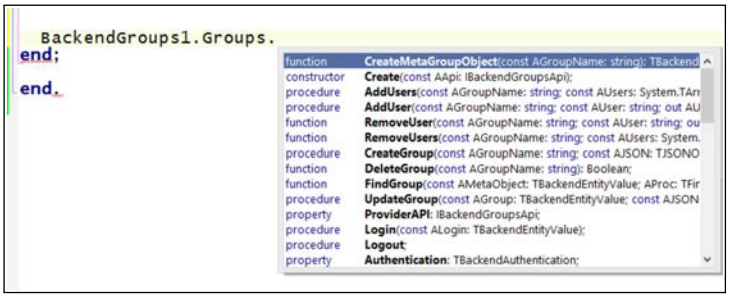


Figura 16.18 MÉTODOS PARA OPERAR SOBRE LA API DE GRUPOS DEL SERVIDOR EMS

- `TBackendQuery`: Facilita la ejecución de cualquier consulta sobre la API base de EMS. El tipo de petición siempre será `GET`. El servicio a invocar lo establece la propiedad `BackendService`. Tras ejecutar la consulta, con una llamada al método `Execute`, se recupera el resultado de la propiedad `JSONResult`. En este caso no existe un procesamiento automático de la respuesta, ya que es un componente genérico para acceder a cualquiera de los servicios.

- `TBackEndPoint`: Permite la ejecución de cualquier *endpoint* existente en el servidor, especificando el tipo de solicitud (propiedad `Method`), el recurso al que se accederá (propiedad `Resource`) y los parámetros a entregar (propiedad `Params`). La llamada puede ser síncrona o asíncrona (método

ExecuteAsync). La respuesta se obtendrá en la propiedad Response, un objeto de tipo TCustomRESTResponse.

- TBackendAuth: Los componentes anteriores, con excepción de TEMSProvider, cuentan con una propiedad Auth que les permite enlazarse con este componente. Su finalidad es configurar los parámetros de autenticación frente al servidor EMS. El tipo de autenticación por defecto, establecida por la propiedad Authentication, usará el contenido de las propiedades UserName y Password para iniciar sesión.
- TEMSClientAPI: Este componente, introducido a partir de la versión XE8 de Delphi, está definido en el módulo REST.Backend.EMSApi. Puede sustituir la funcionalidad de prácticamente todos los anteriores, ofreciendo métodos para obtener las colecciones de grupos y usuarios, crearlos, modificarlos, iniciar sesión, etc.

Veamos en la práctica cómo usar alguno de estos componentes para enumerar los grupos y usuarios existentes, así como para facilitar la creación de nuevos grupos. A partir de estas operaciones básicas es fácil deducir cómo se realizan el resto.

Partiendo de un nuevo proyecto de tipo multi-dispositivo seguiremos el procedimiento indicado a continuación:

1. Introducimos en el formulario un componente TEMSProvider, ajustando sus propiedades URLHost y URLPort para poder acceder al servidor EMS que tenemos en funcionamiento.
2. Agregamos un TBackendQuery que, automáticamente, se enlazará con el componente anterior. Este nos servirá para obtener las listas de grupos y usuarios, tarea que no facilitan los componentes TBackendGroups y TBackendUsers.
3. A continuación añadimos un TBackendGroups. Este también se enlazará con el TEMSProvider. Lo usaremos para permitir al usuario crear nuevos grupos.
4. Usaremos el método OnShow del formulario para llamar a dos métodos privados que tenemos que añadir a la clase. El código del método asociado a dicho evento será el siguiente:

```
procedure TForm12.FormShow(Sender: TObject);
begin
  RefreshGroups;
  RefreshUsers;
end;
```

Listado 16.1 Al mostrar el formulario se actualizarán las listas

5. Implementamos el método `RefreshGroups`. Este utilizará el componente `TBackendQuery` para obtener la lista de grupos y agregarla a un control `TListBox` que forma parte de la interfaz de usuario. No tenemos más que especificar el servicio al que queremos acceder, mediante la propiedad `BackendService`, y llamar al método `Execute`. A continuación extraemos los elementos devueltos en la propiedad `JSONResult`, tal y como se muestra en el siguiente código:

```
...
procedure TForm12.RefreshGroups;
var
  I: Integer;
begin
  lbGroups.Clear;

  BackendQuery1.BackendService := 'Groups';
  BackendQuery1.Execute;
  with BackendQuery1.JSONResult do
  for I := 0 to Count-1 do
    lbGroups.Items.Add(Items[I].GetValue<String>('groupname'));
end;
```

Listado 16.2 Implementación del método `RefreshGroups`

6. Hacemos lo mismo con el método `RefreshUsers`, cuyo código es parecido al anterior, usando el mismo componente `TBackendQuery`:

```
...
procedure TForm12.RefreshUsers;
var
  I: Integer;
begin
  BackendQuery1.BackendService := 'Users';
  BackendQuery1.Execute;
  with BackendQuery1.JSONResult do
  for I := 0 to Count-1 do
    lbUsers.Items.Add(Items[I].GetValue<String>('username
        ↳ '));
end;
```

Listado 16.3 Implementación del método `RefreshUsers`

7. Además de los dos controles `TListBox`, también colocaremos en el formulario un `TEdit` y un `TButton`. Su finalidad será permitir que el usuario agregue nuevos grupos. Hacemos doble clic sobre el botón e introducimos el código necesario, usando en este caso el componente `TBackendGroups`:

```
procedure TForm12.btnGrupoClick(Sender: TObject);
var
  newGroup: TBackendEntityValue;
begin
  BackendGroups1.Groups.CreateGroup(
     edlGrupo.Text, TJSONObject.Create, newGroup);
  RefreshGroups;
end;
```

Listado 16.4 Creación de un nuevo grupo

Asumiendo que el servidor EMS está en funcionamiento, al ejecutar este proyecto veríamos aparecer en las listas los grupos y usuarios previamente registrados. En la Figura 16.19 puede verse cómo se ha añadido un nuevo grupo. Esta acción puede comprobarse desde la consola de desarrollo de EMS.

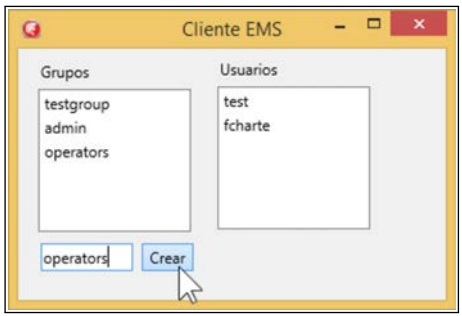


Figura 16.19 EL CLIENTE EMS EN FUNCIONAMIENTO

EJEMPLO 16.1 `EMSClient.dproj`

Puedes encontrar este proyecto completo en la carpeta `EMSClient`. Para que la aplicación funcione correctamente es necesario que esté en funcionamiento el servidor EMS de desarrollo.

16.5 Desarrollo de paquetes EMS

Los servicios que ofrece el servidor EMS *de serie*, sin que tengamos que hacer nada, tienen un objetivo: ahorrarnos todo el trabajo que conllevaría implementarlos en cada servidor REST que necesitemos desarrollar. Para que EMS sea útil, no obstante, tiene que existir un mecanismo que permite integrar en él nuestros propios servicios, con la finalidad específica que nos interese.

EMS está preparado para extender sus capacidades mediante la integración de paquetes. Estos se desarrollan con Delphi, registrando nuevos recursos y los métodos (*endpoint*) asociados para manipularlos. En esta última sección del capítulo se describe el proceso a seguir para desarrollar este tipo de paquetes.

16.5.1 El asistente para paquetes EMS

En la Figura 16.3 (véase la página 473) se mostró el contenido de la rama **EMS** del cuadro de diálogo **NEW ITEMS**. Allí encontraremos el asistente para creación de paquetes EMS, asociado al elemento **EMS PACKAGE**. Este tipo de paquetes se componen de módulos EMS, generados por el segundo elemento, cada uno de los cuales expondría un recurso.

El asistente de paquetes genera un nuevo proyecto. Consta de los siguientes pasos:

1. En el primer paso podremos elegir entre crear un paquete vacío o bien conteniendo inicialmente un módulo con un recurso. Si optásemos por un paquete vacío el asistente finalizaría.

2. Asumiendo que elegimos la opción de incluir un recurso, a continuación elegiremos cuál será la clase ascendiente de la que se derivará el recurso. Podemos elegir entre un módulo de datos, si vamos a acceder a una base de datos, o un módulo normal.

3. En el último paso el asistente nos permite elegir los *endpoints* que queremos incorporar a nuestro paquete EMS. Estos son los mismos (véase la Figura 16.20) que ya conocemos: obtención de la colección, de un elemento, inserción, modificación y borrado.

Al finalizar el asistente tendremos un proyecto que genera un paquete `.bpl`, en lugar de un ejecutable, conteniendo un módulo con la definición de una clase que representa el recurso EMS.

16.5.2 Atributos y registro de las clases de recurso

Una clase que va a actuar como recurso EMS ha de estar *marcada* como tal y ha de autoregistrarse para que el servidor EMS pueda usarla. Para cumplir con el primer requisito utilizaremos atributos específicos, encargados de aportar la información que el servidor EMS necesita. Para la segunda tarea basta con llamar a una función.

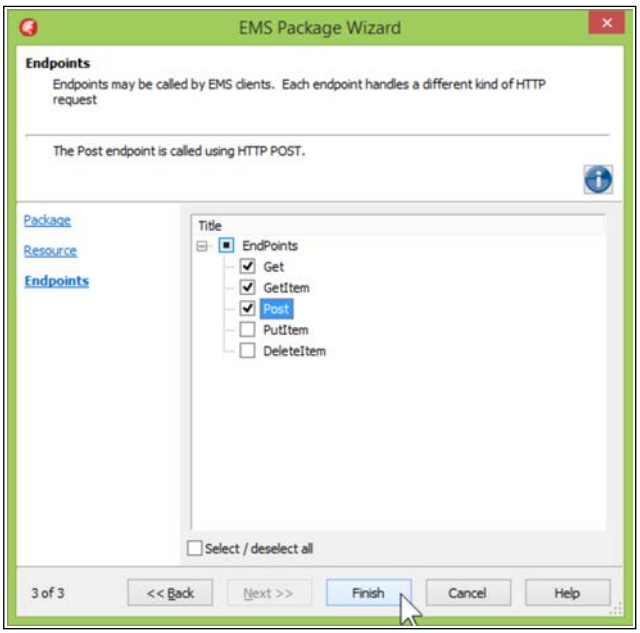


Figura 16.20 ÚLTIMO PASO DEL ASISTENTE PARA PAQUETES EMS

Al definir una clase que va a actuar como recurso EMS deberemos incluir el atributo `ResourceName`. Este precederá a la propia definición, irá entre corchetes y tomará como parámetro el nombre que se asignará al recurso. Por ejemplo:

```
type
  [ResourceName('Rand')]
  TRandResource1 = class(TDataModule)
...
```

Listado 16.5 Atributo que marca una clase como recurso EMS

Esta clase aparecería en el servidor EMS como un recurso con el nombre `Rand`, lo que implica que para acceder a sus servicios se usará un URL del

tipo `http://servidor/Rand`. La clase contará con tantos métodos como *endpoints* vaya a tener el recurso. Lo habitual es que existan los cinco básicos que ya conocemos. Al definir estos métodos hemos de tener presentes varios aspectos:

1. En caso de que a la ruta base haya que agregar un sufijo, a fin de agrupar los métodos o agregar otros nuevos, este habrá de indicarse mediante el atributo `ResourceSuffix`.

2. Si el método toma parámetros en el URL, aparte de los que pudiesen incluir en cabeceras o cuerpo de la solicitud, estos se especificarán con el mismo atributo anterior, facilitando su nombre entre llaves.

3. El prefijo del nombre del método indica el método HTTP que lo invocará. Si el método tiene el sufijo de acceso `Random` y es accesible con solicitud `GET`, el nombre del método Delphi será `GetRandom`.

4. Todos los métodos del recurso toman la misma lista de tres argumentos: `AContext`, `ARequest` y `AResponse`. El primero representa el contexto EMS, el segundo contendrá la información de la solicitud y el tercero permite preparar la respuesta a devolver.

En cuanto al registro como recurso EMS de la clase definida en el módulo, habrá que agregar a este una sección `initialization` desde la que se llame a la función `RegisterResource` facilitando como parámetro la información de tipo de la clase. Esta se extrae con la función `TypeInfo`.

16.5.3 Versión EMS del servidor de números aleatorios

Suponiendo que hemos usado el asistente para crear un nuevo proyecto de paquete EMS con un recurso básico, veamos cómo completaríamos el módulo para tener una versión de nuestro generador de números aleatorios integrable en el servidor EMS.

Lo primero que haríamos sería modificar la definición de la clase, agregando los atributos y firma del método en cuestión. Este será accesible mediante el sufijo `Random`, que se agrega al nombre del recurso. Además necesita dos

parámetros, cuyos nombres también se incluirán en el atributo `ResourceSuffix` como se indicó antes, por lo que la definición quedaría así:

```
...
  published
    [ResourceSuffix('Random/{FromN}/{ToN}')]
    procedure GetRandom(const AContext: TEndpointContext; ↙
        ↳ const ARequest: TEndpointRequest; const AResponse:↙
        ↳  TEndpointResponse);
...
```

Listado 16.6 Definición del método que actuará como *endpoint*

El objeto `TEndpointRequest` recibido como segundo parámetro facilita toda la información relativa a la solicitud, almacenada en propiedades como `Params`, `Headers` y `Body`. Usaremos la primera para extraer los argumentos de entrada. El tercer parámetro, de tipo `TEndpointResponse`, también cuenta con una propiedad `Body`. Esta servirá para introducir la respuesta a devolver al cliente. La implementación de nuestro método quedaría como se muestra a continuación:

```
procedure TRandResource1.GetRandom(
  const AContext: TEndpointContext;
  const ARequest: TEndpointRequest;
  const AResponse: TEndpointResponse);
var
  FromN, ToN: Integer;
begin
  FromN := StrToInt(ARequest.Params.Values['FromN']);
  ToN := StrToInt(ARequest.Params.Values['ToN']);

  AResponse.Body.SetValue(
    TJSONNumber.Create(Random(ToN-FromN) + FromN), True);
end;
```

Listado 16.7 Implementación del método definido antes

De manera análoga implementaríamos el método encargado de cambiar la semilla del generador de números aleatorios. Finalizado el proyecto, podemos ejecutarlo directamente ya que está configurado para lanzar el servidor EMS de desarrollo. En el registro de este podemos comprobar si aparece el recurso `Rand` y, entre sus *endpoints*, el método `GetRandom` (véase la Figura 16.21). Para probar el funcionamiento del servicio podemos servirnos de cualquiera de las herramientas usadas a lo largo del capítulo: un navegador web, el depurador REST de Delphi o los componentes específicos para EMS.

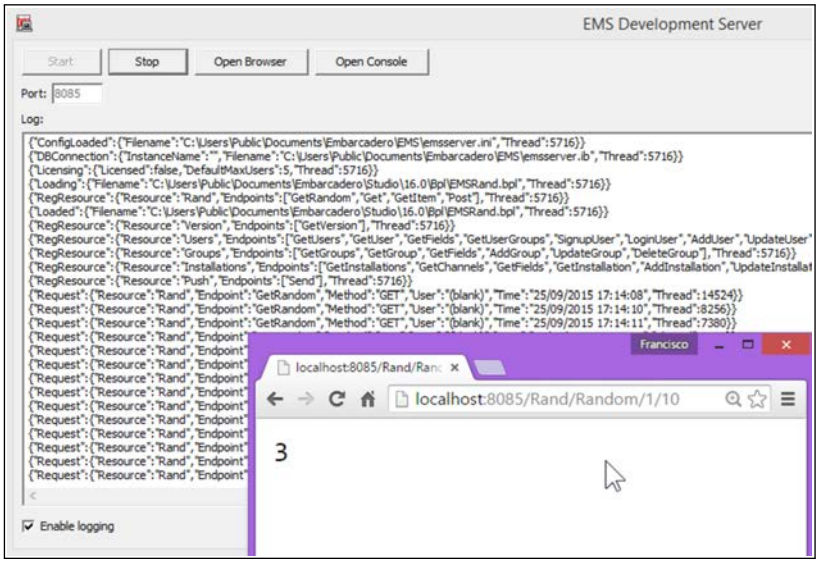


Figura 16.21 PROBAMOS EL SERVIDOR EMS DE NÚMEROS ALEATORIOS

NOTA

La versión final de un paquete a poner en explotación se registraría en el servidor EMS definitivo agregando una entrada al correspondiente archivo de configuración.

16.6 Resumen

En este capítulo hemos conocido una de las herramientas más interesantes inclui-das en la últimas versiones de Delphi: EMS. Este es un servidor REST que implementa servicios básicos de gestión de usuarios, grupos y registro de clientes, funcionando como una base sobre la que podemos conectar la funcionalidad que nos interese ofrecer.

Gracias a EMS el desarrollo de soluciones distribuidas se simplifica de forma considerable, al no tener que ocuparnos de tareas como las citadas, el proceso de autenticación de usuarios y autorización de acceso a los servicios. El servidor EMS incluye, además, una licencia de InterBase que nos permite utilizar este RDBMS para almacenar la información de nuestra aplicación.

PARTE 4

APÉNDICES

Aunque el objetivo principal de este libro es el estudio de los procedimientos a seguir para el desarrollo de aplicaciones con acceso a bases de datos, también se ha intentado que, en la medida de lo posible, el lector encuentre en él toda la información necesaria para trabajar con Delphi, sin necesidad de recurrir a otras fuentes de información. Esta es la razón de ser de los apéndices siguientes:

A. **El entorno de Delphi**: Describe someramente el entorno de desarrollo integrado (IDE) de Delphi, con especial énfasis en aquellas herramientas no relacionadas con el trabajo con bases de datos y que, por ello, no han sido descritas en los capítulos del libro.

B. **El lenguaje Delphi**: Para entender los ejercicios desarrollados en los capítulos de este libro es preciso conocer los fundamentos del lenguaje Delphi. Este apéndice introduce los principales aspectos del lenguaje Delphi, pero sin entrar en conceptos básicos de orientación a objetos, patrones y otras nociones generales.

C. **Integración de Delphi con GIT**: Indistintamente de que trabajemos en un proyecto en grupo o de manera individual, contar con un sistema de control de versiones de código fuente es prácticamente indispensable. **Git** (`http://git-scm.com/`) es un sistema de control de versiones distribuido y *open source* fácilmente integrable con Delphi. En este capítulo se facilitan los fundamentos de trabajo con Git y su uso desde Delphi.

D. **Migración de aplicaciones BDE a FireDAC**: Delphi es un entorno de desarrollo con una larga historia detrás, habiéndose utilizado en la creación de miles de aplicaciones. En muchas de ellas el mecanismo de acceso a datos empleado era BDE (*Borland Database Engine*), un método obsoleto desde hace años. El objetivo de este apéndice es ofrecer una serie de pautas a seguir para actualizar ese tipo de proyectos, sustituyendo BDE por FireDAC.

APÉNDICE A
EL ENTORNO DE DELPHI

Los entornos de desarrollo de aplicaciones utilizados en la actualidad son considerablemente complejos, con independencia de los lenguajes de programación que soporten o los sistemas operativos y plataformas para las que se genere software. En este sentido el entorno de Delphi no es una excepción, contando con innumerables opciones repartidas por distintos menús y paneles, barras de herramientas, etc. Para sacar provecho de toda esta funcionalidad es importante conocer a fondo dicho entorno, una tarea que habitualmente requerirá muchas horas de trabajo por parte los desarrolladores.

El objetivo principal de este libro es ofrecer al usuario toda la información necesaria para que pueda desarrollar aplicaciones de bases de datos con Delphi. Por ello este apéndice es una parte fundamental para aquellos usuarios que no conocen el entorno actual de Delphi, ya que el primer paso que habrán de dar es familiarizarse con dicho entorno a fin de poder seguir las indicaciones dadas en cada capítulo. Si eres un usuario habitual de Delphi el contenido de este apéndice puede servirte como material de referencia, para encontrar la opción o herramienta a la que has de recurrir para realizar una tarea concreta. No obstante, en las páginas siguientes no se enumeran exhaustivamente todas las opciones ofrecidas por Delphi, este es un material que podemos encontrar en la documentación electrónica del producto, sino que se guía al lector de forma didáctica por las distintas tareas a completar.

A.1 El IDE del Delphi

Para usar el entorno de desarrollo o IDE (*Integrated Development Environment*) de Delphi tendremos que comenzar por conocer los principales elementos que lo forman, su denominación y localización. De esta forma sabremos dónde recurrir a la hora de iniciar un nuevo proyecto, administrar sus componentes, compilar el proyecto, etc.

En la Figura A.1 se indica la posición por defecto en la que podemos encontrar los elementos del entorno de Delphi enumerados a continuación. Esta denominación es la que se utiliza principalmente a lo largo de este libro:

- **Menú principal** y **Barra principal de herramientas**: Están situados en la parte superior de la ventana principal del IDE. Son los mecanismos que facilitan el acceso a los comandos más comunes, en el caso de la barra de herramientas, y a la mayor parte de las opciones de Delphi, en el caso del menú. Al igual que muchos otros componentes del IDE, la barra de herramientas puede personalizarse agregando y quitando botones y otras herramientas.
- **Pestañas de módulos abiertos**: El entorno puede tener abiertos múltiples módulos de forma simultánea, cada uno de ellos en respectivo diseñador o editor. Las pestañas situados justo debajo de la barra de herramientas muestra el nombre de cada módulo abierto y permiten alternar entre ellos.

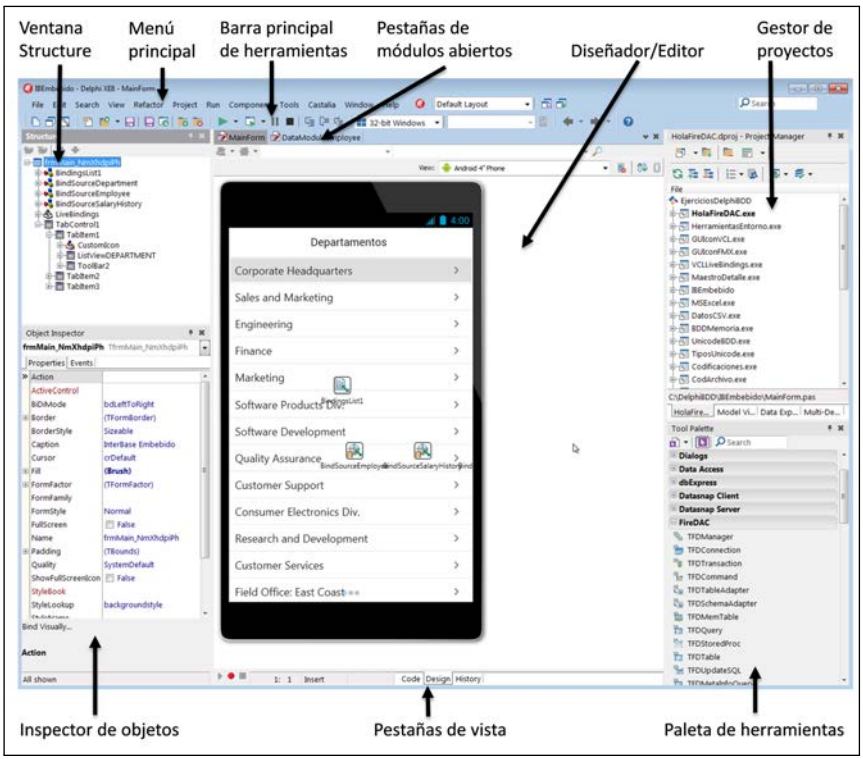


Figura A.1 ELEMENTOS FUNDAMENTALES DEL IDE DE DELPHI

- **Diseñador/Editor**: Al abrir cualquier módulo de un proyecto, sencillamente haciendo doble clic sobre su nombre en el Gestor de proyectos, se recurrirá a un editor o un diseñador, dependiendo del tipo de módulo, que ocupará el área central del entorno de trabajo. Las dos herramientas más habituales son el diseñador de interfaces de usuario, que puede apreciarse en la Figura A.1, y el editor de código.

- **Pestañas de vista**: Situadas en la parte inferior del módulo abierto en el área central, con estas pestañas podemos alternar entre las distintas vistas asociadas a dicho módulo. Un formulario, por ejemplo, cuenta con las vistas

CODE y **DESIGN**, correspondientes al editor de código y el diseñador de interfaces, respectivamente.

- **Gestor de proyectos**: Este panel contendrá el grupo de proyectos en el que estemos trabajando. Cada proyecto será un nodo con múltiples hojas, cada una de las cuales representa un módulo, un conjunto de parámetros de configuración, la lista de plataformas para las que se generará el proyecto, etc.
- **Paleta de herramientas**: El contenido de este panel es sensible al contexto, ofreciendo opciones para creación de nuevos proyectos o bien los componentes que pueden alojarse en un contenedor cuando se tiene abierto un diseñador.
- **Inspector de objetos**: Una vez que se agrega a un contenedor un componente, tomado de la Paleta de herramientas, podremos usar el Inspector de objetos para personalizar su apariencia y comportamiento, modificando el valor de las propiedades y asociando código a los eventos que genera.
- **Ventana Structure**: La finalidad de este panel es ofrecer una visión general de la estructura del módulo que se tiene abierto. Al trabajar con un diseñador el contenedor principal, por ejemplo el formulario, actuará como nodo raíz, del cual irán colgando los distintos componentes insertados en él. Cuando se trabaja con el editor de código este panel permite navegar rápidamente por los miembros de las clases que contiene.

A.1.1 Cómo cambiar la distribución de los paneles del IDE

Aunque en principio la distribución de los diferentes paneles del IDE coincidirá con la mostrada antes en la Figura A.1, lo cierto es que todos ellos pueden abrirse, cerrarse, ocultarse de forma automática, separarse de la ventana principal del IDE quedando como ventanas flotantes, apilarse en grupos que actúan como páginas, etc. Los elementos que nos permiten cambiar la distribución y comportamiento de los paneles son las señaladas en la Figura A.2. Las acciones que podemos llevar a cabo sobre estos elementos son las siguientes:

- **Acoplar/Desacoplar**: Con un clic sobre la barra de título de una ventana y arrastrando y soltando podemos tanto desacoplarla de su localización ac-

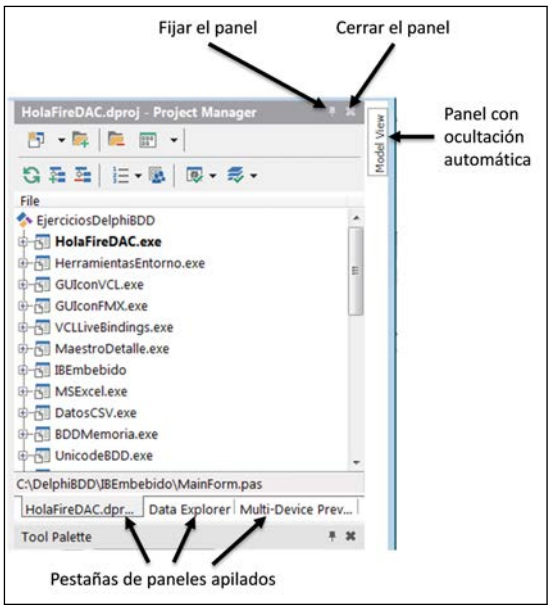


Figura A.2 HERRAMIENTAS DE PERSONALIZACIÓN DE LOS PANELES

tual, dejándola como flotante, como acoplarla a otro de los márgenes de la ventana principal del IDE. También es posible apilar un panel sobre otro para que ocupen el mismo espacio, apareciendo como páginas de una misma ventana.

- **Desapilar**: Si queremos desapilar uno de los paneles que aparece como una página de una ventana usaremos la misma técnica anterior, pero haciendo clic sobre la pestaña correspondiente en lugar de hacerlo sobre la barra de título. De esta forma podremos situar el panel individual donde nos interese.
- **Ocultar automáticamente**: Con excepción del área en el que se abre el editor/diseñador, el resto de ventanas existentes en el IDE pueden ser ocultadas de forma que solo se muestre de ellas una pestaña con el título en uno de los márgenes de la ventana principal. Esto se aplica tanto a paneles individuales como a ventanas con múltiples paneles apilados. Haciendo clic en el botón en forma de chincheta se alterna entre el estado normal de la ventana y la

ocultación. Una ventana con este modo activado se abre automáticamente en cuanto se sitúa el puntero del ratón sobre la pestaña correspondiente, volviendo a cerrarse cuando ya no es necesaria.

- **Cerrar/Abrir**: El botón situado en la esquina superior derecha de cada ventana, en forma de aspa, es el encargado de cerrarla, incluyendo todos los paneles que contuviese. Para volver a abrir un panel no hay más que usar la opción correspondiente del menú **VIEW**.

Podemos contar con múltiples configuraciones del IDE configuradas y guardadas, alternando entre ellas cómodamente mediante la lista desplegable situada en el extremo derecho de la barra del menú principal. Para crear una nueva configuración tendríamos que seguir este procedimiento:

1. Comenzaríamos ajustando la distribución de todos los elementos del IDE, estableciendo la posición y estado de cada uno de los paneles.
2. Hacemos clic en el botón **SAVE CURRENT DESKTOP**[1] tal y como se muestra en la Figura A.3.
3. Introducimos el nombre con el que deseamos conservar la nueva configuración.

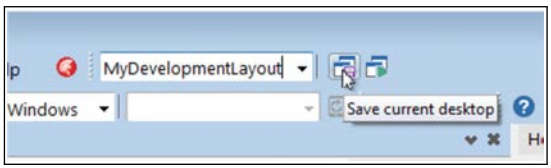


Figura A.3 GUARDAMOS LA CONFIGURACIÓN ACTUAL DEL IDE

Tras haber definido la nueva configuración podremos alternar entre esta y cualquier otra seleccionando su nombre de la lista desplegable. De esta forma podríamos, por ejemplo, establecer una distribución para cuando vayamos a centrarnos en el diseño de interfaces de usuario y otra para la edición de código.

[1] Junto a este botón encontramos otro llamado **SET DEBUG DESKTOP** que permite guardar una distribución de escritorio alternativa que se activaría durante las sesiones de depuración.

A.1.2 La herramienta IDE Insight

La cantidad de opciones, herramientas y componentes disponibles en el entorno de trabajo de Delphi es tan extenso que, en ocasiones, puede resultar tedioso localizar aquella que necesitamos utilizar en cada momento. Siempre podemos recurrir, sin embargo, a IDE Insight, una herramienta capaz de buscar objetos en todo el IDE de Delphi, incluyendo los proyectos que tengamos en ese momento abiertos en el Gestor de proyectos.

La vía más rápida para acceder a IDE Insight es el atajo de teclado **CONTROL +.**. Alternativamente podemos hacer clic sobre el recuadro de búsqueda situado en la esquina superior derecha de la ventana principal del entorno o bien usar la opción **SEARCH—IDE INSIGHT**. En cuanto introduzcamos los primeros caracteres aparecerá una lista (véase la Figura A.4) con comandos, componentes, opciones y módulos de proyectos que los contengan.

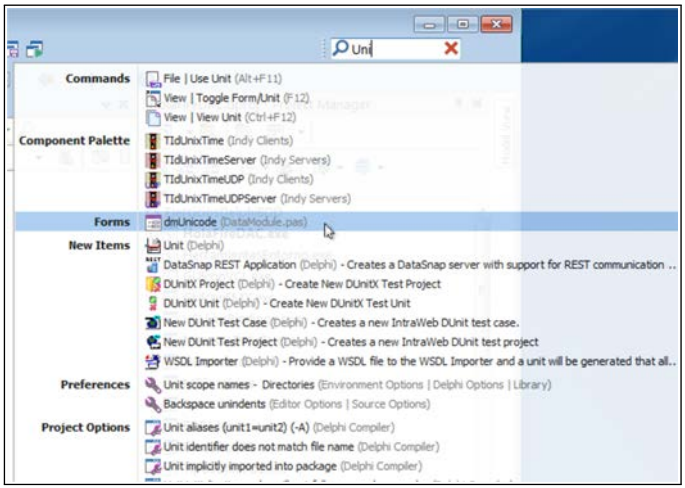


Figura A.4 BÚSQUEDA CON IDE INSIGHT

Dependiendo de la categoría de cada elemento, al elegirlo, por ejemplo haciendo clic sobre él, se ejecutará el comando indicado, se abrirá el módulo correspondiente, se agregará al proyecto el tipo de objeto seleccionado, etc. IDE Insight, por tanto, representa un atajo para acceder al menú de opciones, la Paleta

de componentes, el Gestor de proyectos y la barra de herramientas. Lo único que necesitamos es conocer el nombre del elemento que nos interesa.

A.2 Gestión de proyectos

Para crear una aplicación Delphi, independientemente de cuál sea su tipo y la biblioteca de componentes empleada, siempre tendremos que partir de un proyecto. Esta es la entidad encargada de agrupar los módulos a partir de los que se generará el programa, así como todas las opciones que afectan a la generación del mismo.

El primer paso será, por tanto, crear un nuevo proyecto, usando para ello una de las plantillas que nos ofrece el entorno. A continuación usaríamos el Gestor de proyectos para abrir, agregar y eliminar módulos, modificar las opciones que afectan al proceso de compilación, las plataformas objetivo, etc. Estas son las tareas descritas en los apartados de la presente sección.

A.2.1 Proyectos y plantillas

Delphi nos permite crear múltiples tipos de proyecto, desde aplicaciones multi-dispositivo a módulos de servidor y servicios web, pasando por componentes y bibliotecas de servicios. Para ciertos tipos de proyecto es posible elegir entre un conjunto de plantillas, de forma que se agreguen al nuevo proyecto ciertos elementos predefinidos.

Suponiendo que quisiésemos iniciar un nuevo proyecto de tipo multi-dispositivo[2], podríamos seguir cualquiera de los siguientes procedimientos:

- Elegir la opción **FILE—NEW—MULTI-DEVICE APPLICATION** o bien la opción **FILE—NEW—OTHER** para abrir el cuadro de diálogo **NEW ITEM** (v éase la Figura A.5) eligiendo del mismo la opción **MULTI-DEVICE APPLICATION**.

[2]Esta elección implica utilizar la biblioteca de componentes FireMonkey (FMX). La opción **VCL FORMS APPLICATION** generará un proyecto basado en la VCL, biblioteca disponible únicamente para Windows.

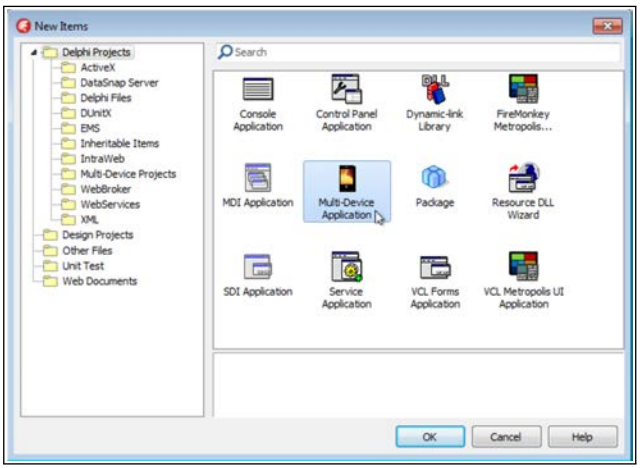


Figura A.5 TIPOS DE PROYECTO DISPONIBLES

- Usar IDE Insight con el término de búsqueda `multi`, eligiendo de la lista de resultados (véase la Figura A.6) la opción **MULTI-DEVICE APPLICATION**.

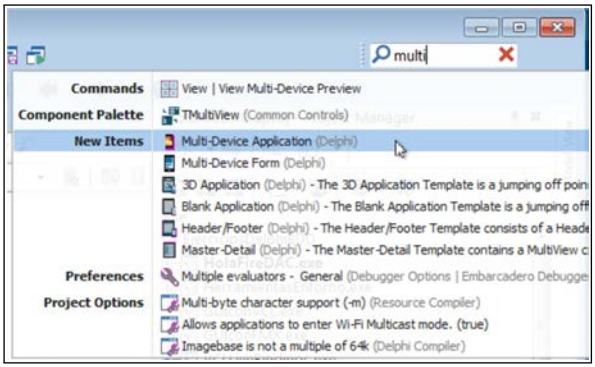


Figura A.6 BÚSQUEDA DE LA OPCIÓN CON IDE INSIGHT

- Recurrir a la Paleta de herramientas, abriendo el grupo **DELPHI PROJECTS —MULTI-DEVICE PROJECTS** (véase la Figura A.7) para elegir directamente la plantilla a partir de la que se creará el nuevo proyecto.

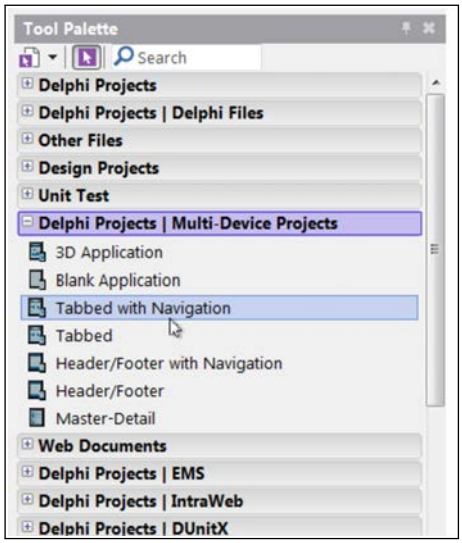


Figura A.7 PROYECTOS Y PLANTILLAS EN LA PALETA DE HERRAMIENTAS

Salvo en el último caso, en el que podemos elegir directamente una plantilla de proyecto a partir de la cual se crearía el nuevo, la opción elegida dará paso a una ventana como la de la Figura A.8. En esta se enumeran las plantillas de proyecto disponibles para la categoría elegida, en este caso las correspondientes a un proyecto multidispositivo.

Al elegir cualquiera de las plantillas obtendremos en la parte inferior una descripción de su finalidad. Básicamente tenemos un proyecto blanco, otro para aplicaciones 3D y combinaciones en las que se agrega al formulario del nuevo proyecto componentes que actúan como cabecera y pie, que permiten contar con múltiples páginas en la interfaz y que facilitan el uso de estas paginas para mostrar una relación maestro/detalle.

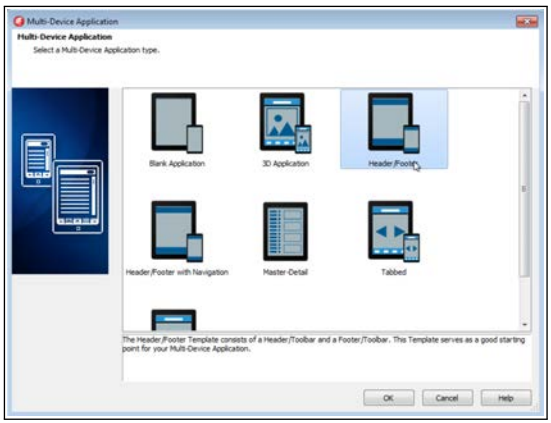


Figura A.8 PLANTILLAS DE PROYECTO MULTIDISPOSITIVO

> **NOTA**
>
> *En versiones de Delphi previas a la XE7 no existe la categoría* **MULTI-DEVICE APPLICATION**, *debiendo crearse por separado proyectos para equipos de escritorio y proyectos para dispositivos móviles.*

A.2.2 El Gestor de proyectos

Tras crear un nuevo proyecto, usando algunas de las opciones anteriormente indicadas, aparecerá en el Gestor de proyectos como un nodo con varias ramas. Cada una de estas puede tener uno o más módulos, dependiendo de su tipo, o bien representar un conjunto de opciones. Cada uno de los nodos, tanto el raíz que representa al proyecto como las ramas y hojas finales, cuentan con un menú contextual específico que facilita las operaciones disponibles para cada tipo de objeto.

Para abrir un módulo, ya sea un formulario, un módulo de código o de otro tipo, no tenemos más que hacer doble clic sobre el nodo que lo representa (véase la Figura A.9) en el Gestor de proyectos. El IDE determinará qué diseñador o

editor es el adecuado, abriéndolo en el área central de la ventana principal del entorno. Tanto formularios como módulos de datos aparecerán en el Gestor de proyecto como módulos con extensión `.pas` que tienen como hijo un segundo módulo, con extensión `.fmx` y `.dfm` respectivamente. Estos últimos almacenan la información relativa a los componentes introducidos en el formulario o módulo de datos y los valores asignados a sus propiedades.

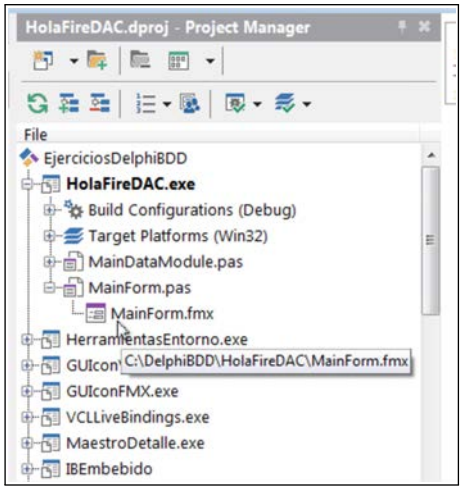


Figura A.9 EL GESTOR DE PROYECTOS

> **NOTA**
>
> *Inicialmente los módulos y proyectos aparecerán en el Gestor de proyecto con un nombre por defecto que podremos cambiar al guardarlos.*

El Gestor de proyectos nos permite mantener múltiples proyectos de forma simultánea, agrupados bajo un nodo que representa el grupo de proyectos. Esto resulta útil, por ejemplo, al desarrollar aplicaciones cliente/servidor o distribuidas, manteniendo como un grupo los proyectos correspondientes a cada parte de la solución. Para añadir un nuevo proyecto al grupo actual no usaríamos las opciones anteriormente descritas, ya que estas cerrarán el grupo actualmente

abierto y generarán uno nuevo, sino que usaríamos la opción **ADD NEW PROJECT** del menú contextual del grupo de proyectos, como se muestra en la Figura A.10.

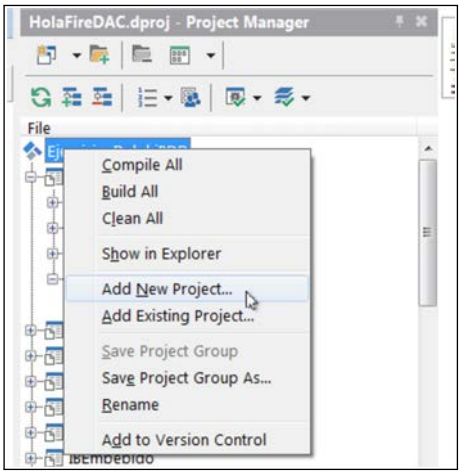


Figura A.10 AGREGAMOS UN NUEVO PROYECTO AL GRUPO ACTUAL

A.3 Vistas de diseño

Cuando se trabaja en la interfaz de un proyecto multi-dispositivo, que presumiblemente se ejecutará en pantallas con distintas resoluciones y orientaciones, el ajuste del diseño para cada caso concreto es una parte importante del desarrollo. Afortunadamente el IDE de Delphi incorpora todas las herramientas necesarias para facilitar esta tarea.

En esta sección se describe el procedimiento a seguir para cambiar de vista durante la fase de diseño, personalizar la vista para determinados tipos de dispositivo, obtener una vista previa de cada caso considerado, etc. Se asume que la versión de Delphi utilizada es la XE7 o posterior, ya que en versiones previas las opciones que van a describirse no están disponibles. Algunos elementos mencionados aparecieron en la versión XE8.

A.3.1 Cambiar la vista activa

En la parte superior del área de trabajo correspondiente al diseñador encontramos una pequeña barra de herramientas. Esta incluye una lista desplegable llamada **VIEWS**, mediante la cual podemos cambiar la vista activa en cada momento. En ella encontraremos las configuraciones habituales para las distintas plataformas (véase la Figura A.11), incluyendo distintos tamaños de teléfonos y tabletas con iOS y Android.

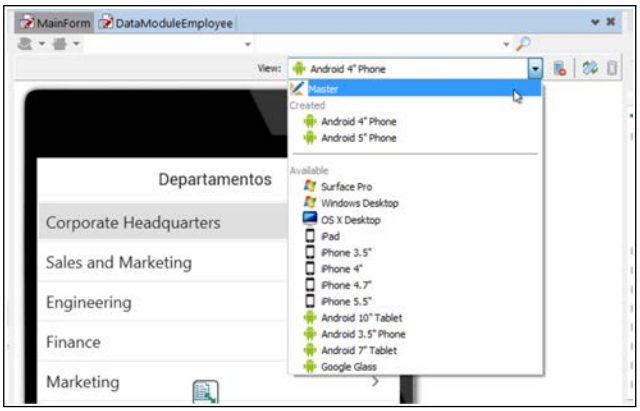


Figura A.11 LISTA CON LAS VISTAS DE DISEÑO DISPONIBLES

La vista **MASTER** es siempre la activa por defecto. El resto se agruparán en dos categorías, dependiendo de que las hayamos activado o no. Al cambiar de vista apreciaremos cómo el diseñador cambia las proporciones del contenedor y también la máscara mostrada alrededor del mismo, simulando el tipo de dispositivo seleccionado.

El botón a la derecha de la lista desplegable nos permite desactivar la vista seleccionada actualmente (siempre que no sea la vista maestra), de forma que pasaría de la categoría de vista creada a la de vistas disponibles.

Inicialmente todas las vistas mostrarán exactamente los mismos componentes, en la misma posición y con el mismo tamaño, si bien según las dimensiones del contenedor y su orientación algunos de ellos podrían ajustarse o quedar fuera del área visible. En un apartado posterior se explica cómo personalizar cada vista.

A.3.2 Orientación y máscara

En la barra de herramientas encontramos, a la derecha de la lista de desplegable de vistas, dos botones adicionales. La finalidad de los mismos es la siguiente:

- **ROTATE THE CURRENT VIEW**: Alterna entre la vista vertical y apaisada, permitiendo comprobar cómo afecta a la interfaz el cambio de orientación del dispositivo. Solo estará activo para vistas correspondientes a tabletas y móviles, no para las configuraciones de escritorio ni la vista maestra.
- **HIDE OR SHOW THE FORM FRAME**: Alrededor del espacio que realmente actúa como contenedor el diseñador puede mostrar una máscara. La finalidad de esta es ofrecer una visualización que simula el tipo de dispositivo seleccionado. Con este botón se muestra y oculta dicha máscara.

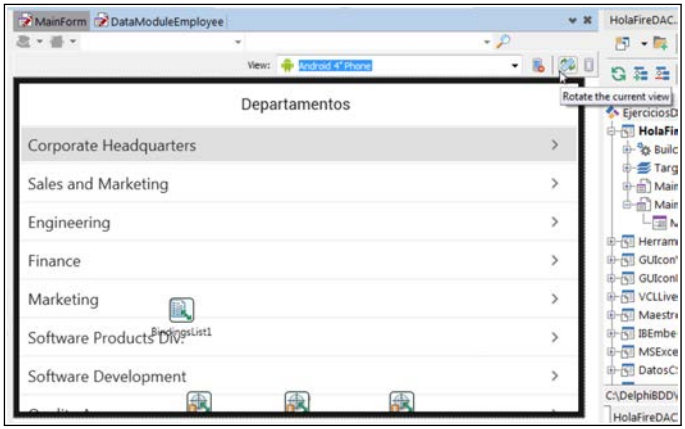


Figura A.12 PODEMOS ROTAR LA VISTA Y DESACTIVAR LA MÁSCARA DE DISPOSITIVO

> **NOTA**
>
> *Si la vista seleccionada es la vista maestra, en la lista desplegable* **STYLE** *que aparece a la izquierda de la anterior podremos elegir el estilo visual con que deseamos mostrar el contenido del formulario.*

A.3.3 Vista maestra y vistas específicas

El diseñador de formularios multi-dispositivo de Delphi cuenta (desde la versión XE7) con un mecanismo que facilita el ajuste de la interfaz de usuario para cada familia de dispositivos. En lugar de componer una interfaz independiente para cada plataforma, se parte de una configuración base representada por la vista maestra. Esta establece los componentes comunes a la interfaz, su estado, posición y dimensiones base. Estos parámetros pueden después diferenciarse para plataformas específicas, activando la vista correspondiente y moviendo los controles, modificando sus propiedades o incluso agregando nuevos componentes si fuese preciso. La mecánica general de trabajo constará de los siguientes pasos:

1. Se parte trabajando en la vista maestra, agregando los componentes comunes a todas las plataformas y estableciendo una configuración general en cuando a disposición de los controles y su estado. Habitualmente sólo las propiedades asociadas a la apariencia, como posición y tamaño, serán las que cambien entre plataformas.

2. A continuación se crea una vista específica para cada plataforma objetivo en la que se vaya a distribuir la aplicación. El efecto inmediato es que podemos comprobar cuál sería el aspecto de la interfaz de usuario en dicha plataforma. Asimismo se pueden realizar los ajustes necesarios en los componentes ya existentes en el formulario. Los cambios que hagamos afectarán únicamente a la vista activa en ese momento, al cambiar a la vista maestra, u otra de las vistas creadas anteriormente, cada una preservará su propia configuración.

3. En caso necesario, es posible agregar nuevos componentes a una vista específica. Estos no aparecerán en el resto de vistas. Por el contrario, cualquier elemento que añadamos a la vista maestra será *heredado* por todas las demás vistas. En caso de que un componente de la vista maestra no sea necesario en una vista específica, podemos usar su propiedad `Visible` para evitar que sea visible.

Por cada vista adicional que creemos se añadirá al proyecto un módulo `.fmx` con la información necesaria para mantenerla. El nombre de estos módulos denota el tipo de dispositivo al que corresponde la vista. El Gestor de proyecto no los mostrará, ya que todos ellos están asociados al módulo `.fmx` principal

que actúa como vista maestra. Si abrimos el módulo de código correspondiente al formulario, sin embargo, encontraremos las directivas de importación para el compilador, tal y como se aprecia en la Figura A.13. La directiva además indica el sistema operativo al que corresponde cada vista, de forma que el compilador incorpore las vistas adecuadas al generar el proyecto para una cierta plataforma.

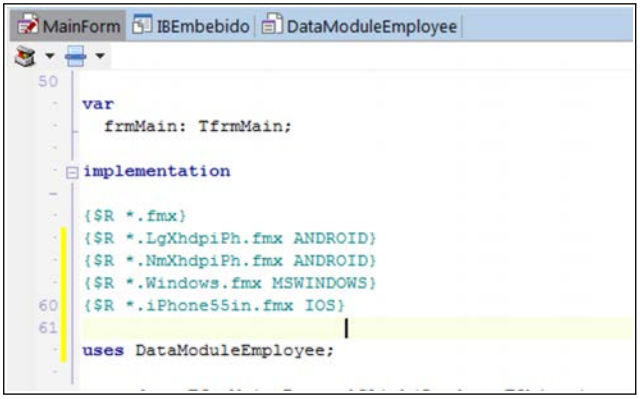


Figura A.13 IMPORTACIÓN DE LOS MÓDULOS DE LAS VISTAS

A.3.4 Visualización preliminar de la interfaz

Aunque el propio diseñador de formularios nos permite obtener una idea básica sobre cómo quedará la interfaz en cada tipo de dispositivo, lo cierto es que solo podemos tener una vista activa en cada momento y, por tanto, no es fácil realizar comparaciones. Además, en el diseñador también son visibles componentes que al ejecutar el proyecto no lo serían, especialmente los relacionados con la conexión a datos.

Desde la versión XE8 el IDE de Delphi incorpora un nuevo panel, llamado **MULTI DEVICE PREVIEW**, que facilita la visualización previa de la interfaz de usuario en tantos tipos de dispositivo distinto como necesitemos. Este panel aparece por defecto apilado con los paneles correspondiente al Gestor de proyectos y el Explorador de datos, pero podemos desacoplarlo y dejarlo como un panel flotante, situándolo donde más cómodo nos resulte. Cuando más espacio demos a dicho panel más útil nos resultará.

La barra de herramientas situada en la parte superior de este panel (véase la Figura A.14) cuenta con dos botones desplegables. Con el primero podremos elegir los dispositivos cuya visualización previa aparecerá en esta ventana. La finalidad del segundo es permitirnos elegir la distribución de las previsualizaciones en el espacio disponible en la ventana. Usando el control deslizador situado en la parte inferior derecha del panel podemos ajustar el tamaño de las previsualizaciones.

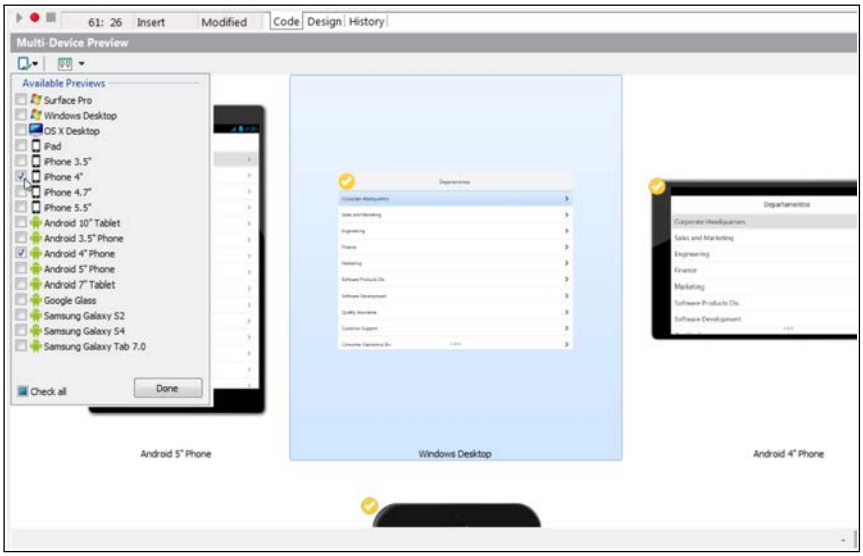


Figura A.14 VISUALIZACIÓN PRELIMINAR PARA CADA PLATAFORMA

El aspecto quizá más interesante de esta herramienta es que la previsualización se actualiza de manera inmediata mientras trabajamos en el diseñador. Esto nos permite ir componiendo la interfaz de usuario al tiempo que comprobamos cuál sería su aspecto en distintos sistemas y tipos de dispositivo. Lógicamente cuanto más espacio podamos dedicar al panel mayor será el tamaño de la previsualizacion, haciendo más cómoda esta verificación.

Las previsualizaciones para las que se haya creado en el diseñador una vista específica, según el procedimiento descrito en el apartado previo, mostrarán en su esquina superior izquierda un icono con una marca de verificación. Para

agregar una nueva vista asociada a un tipo de dispositivo podemos hacer doble clic sobre la previsualización correspondiente, respondiendo afirmativamente a la pregunta de si deseamos crearla (véase la Figura A.15).

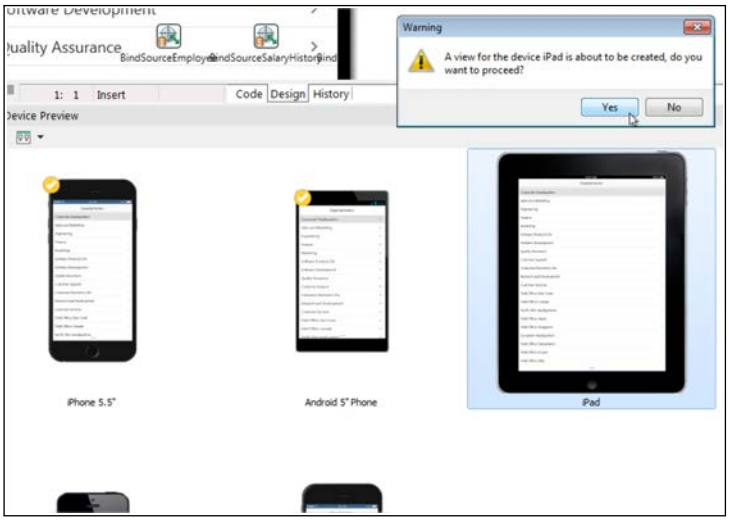


Figura A.15 AGREGAMOS AL DISEÑADOR UNA NUEVA VISTA

También podemos hacer doble clic sobre cualquiera de las previsualizaciones para las que existe una vista a fin de activar esta en el diseñador, como vía alternativa a desplegar la lista **VIEWS** y elegirla.

A.4 Trabajar con componentes

Tras crear un nuevo proyecto a partir de una de las plantillas disponibles, a excepción de la que genera un proyecto vacío, contaremos en el formulario con un grupo de componentes. A estos se podrán agregar otros que necesitemos. Todos ellos contendrán valores por defecto en sus propiedades que, en muchos casos, necesitaremos editar para ajustarlos a nuestras necesidades. En esta sección se describen los procedimientos a seguir para completar cada una de estas tareas.

A.4.1 Adición de nuevos componentes

Con independencia de que estemos diseñando una interfaz de usuario, utilizando como base de partida un formulario; definiendo un modelo de datos a usar en la aplicación, recurriendo a un módulo de datos, o creando algún tipo de servidor con otro tipo de contenedor, lo cierto es que siempre nos encontraremos con una necesidad común: agregar a dicho contenedor los componentes necesarios para implementar la funcionalidad que se espera de la aplicación.

Todos los componentes disponibles para el desarrollo de aplicaciones los encontraremos en la Paleta de herramientas. Los grupos presentes en este panel, y los componentes en cada grupo, dependerán de las bibliotecas que se hayan instalado el IDE. No obstante componentes como FireDAC, FMX y VCL siempre estarán ahí (véase la Figura A.16).

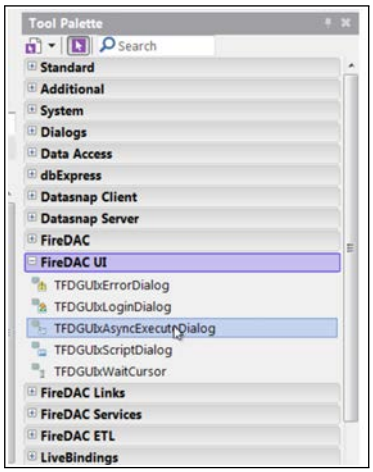


Figura A.16 COMPONENTES EN LA PALETA DE HERRAMIENTAS

El procedimiento para añadir nuevos componentes al contenedor que tengamos abierto en el diseñador es siempre el mismo, constando esencialmente de dos pasos:

1. El primer paso será encontrar el componente que se necesita. Si sabemos la categoría a que pertenece podemos abrir la sección correspondiente para

localizarlo. En caso de que conozcamos su nombre, podemos introducir parte del mismo en el recuadro de búsqueda que hay en la parte superior de la Paleta de herramientas para filtrar el contenido mostrado.

2. Localizado el componente que nos interesa, para agregarlo al contenedor podemos recurrir a distintas estrategias:
   - Con un doble clic sobre el componente se insertará en el contenedor una copia del mismo con una posición y dimensiones por defecto.
   - Haciendo clic sobre el componente y a continuación en el contenedor se insertará una copia en la posición seleccionada.
   - Haciendo clic sobre el componente y a continuación arrastrando y soltando sobre el contenedor para fijar tanto la posición como las dimensiones.

Al insertar un componente hemos de tener en cuenta la relación padre-hijo que tendrá respecto a otros elementos que ya pudieran existir en el contenedor. Ciertos controles de interfaz de usuario, quizá el más básico sea `TPanel`, actúan como contenedores de otros. Dependiendo de cuál sea el objeto seleccionado en el contenedor, o bien del objeto sobre el que hagamos clic para insertar el nuevo componente, se establecerá la relación padre-hijo correspondiente. Siempre podemos comprobar esta relación en la ventana **STRUCTURE** (véase la Figura A.17).

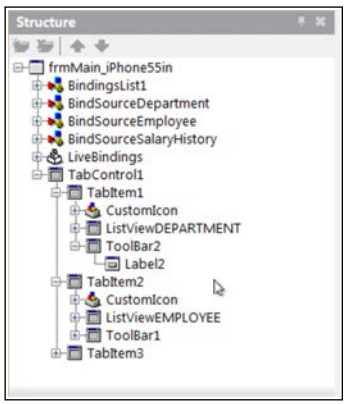


Figura A.17 RELACIÓN ENTRE LOS COMPONENTES ALOJADOS EN EL CONTENEDOR

> **NOTA**
>
> *Si necesitamos insertar varias copias de un mismo componente nos resultará más cómodo hacer clic sobre él en la Paleta de herramientas manteniendo pulsada la tecla* **MAYÚS** *para, a continuación, hacer clic sobre el formulario tantas veces como necesitemos, obteniendo una copia del componente en cada clic.*

A.4.2 Manipulación de componentes

Una vez que los componentes se encuentran ya en su contenedor, para actuar sobre ellos será necesario seleccionarlos, ya sea individual o conjuntamente. El procedimiento de selección es el habitual en la mayoría de aplicaciones Windows. Un clic selecciona el componente situado bajo el puntero, perdiéndose la selección del que estuviese activo en ese momento. Manteniendo pulsada la tecla **MAYÚS** y haciendo clic se selecciona un componente sin perder los ya seleccionados. También se puede arrastrar y soltar sobre el fondo del contenedor, trazando un recuadro, a fin de seleccionar todos aquellos componentes existentes en el área marcada.

Teniendo seleccionado uno o más componentes, algunas de las acciones que podemos llevar a cabo son:

- **Mover**: Mediante la técnica de arrastrar y soltar es posible mover tanto uno como un conjunto de componentes. El clic inicial ha de hacerse en el interior de uno de los componentes seleccionados.
- **Redimensionar**: Si el clic inicial se hace sobre uno de los pequeños círculos que aparecen alrededor de un componente, al arrastrar y soltar no cambiaremos su tamaño sino sus dimensiones. Solamente los controles, componentes con una parte visible en la interfaz, pueden cambiarse de tamaño.
- **Eliminar**: Pulsando la tecla **SUPR** todos los componentes seleccionados serán eliminados del contenedor. Esta acción, al igual que muchas otras de edición, puede deshacerse mediante el habitual comando asociado al atajo **CTRL+Z**.

- **Copiar y pegar**: Cuando se precisan múltiples copias de un componente (o conjunto de componentes) con una configuración similar, resultará más rápido personalizar la primera copia, seleccionarla, copiarla al portapapeles con **CTRL+C** y después pegarla donde se necesite, quizá en otro contenedor, con el atajo **CTRL+V**.

Dependiendo de cómo esté configurado cada componente, seleccionarlo directamente en el contenedor puede resultar en ocasiones difícil. Esto suele ocurrir cuando un control ocupa todo el espacio disponible en un contenedor, por ejemplo, de forma que hagamos donde hagamos clic seleccionados el control, no su contenedor que es lo que podría interesarnos. En estos casos podemos usar la ventana **STRUCTURE** para seleccionar los objetos que nos interesen. Incluso ciertas acciones, como el borrado y el uso del portapapeles, pueden llevarse a cabo directamente en esta ventana.

A.4.3 Edición de propiedades

Si bien ciertas propiedades de un componente pueden manipularse visualmente, por ejemplo al cambiar su posición o dimensiones, el resto no son accesibles de forma directa en el formulario. La finalidad del Inspector de objetos es facilitar la edición de todas las propiedades del componente seleccionado. En caso de que tengamos marcados varios controles simultáneamente, solo tendremos acceso a las propiedades comunes a todos ellos.

El Inspector de objetos cuenta con dos páginas: **PROPERTIES** e **EVENTS**. La abierta por defecto es la primera, conteniendo el nombre y contenido actual de cada una de las propiedades expuestas por el componente. El nombre obviamente es inmutable, sería el que utilizaríamos para acceder a cada propiedad desde el código del proyecto. El contenido puede ser de distintos tipos, dependiendo de lo cual su modificación se llevará a cabo por un procedimiento u otro. Los casos que podemos encontrar son los descritos a continuación:

- **Cadenas de texto y números**: Las propiedades de tipo `String`, `Integer` y otros tipos numéricos muestran su valor actual en un recuadro de texto que permite la edición directa, por lo que no hay más que hacer clic sobre el recuadro para darle el foco de entrada e introducir el nuevo valor (véase la Figura A.18).

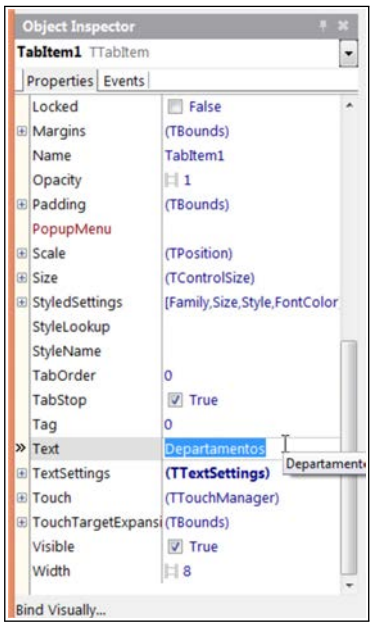


Figura A.18 VISTA PARCIAL DEL INSPECTOR DE OBJETOS

- **Booleanos**: Este tipo de propiedad aparece en el Inspector de objetos como un botón que podemos marcar (`True`) o desmarcar (`False`). Es todo lo necesario para cambiar su contenido.

- **Enumeraciones**: Propiedades como `Align` y `StyleLookup` pueden contener un valor de una enumeración. Los valores permitidos aparecerán en forma de lista desplegable, si bien el recuadro en el que se muestra el valor actual también permite la edición directa siempre que el dato introducido sea válido.

- **Conjuntos**: Estas propiedades pueden tener activos de forma simultánea varios valores de un conjunto predefinido. Muestran a la izquierda de su nombre un botón + que permite desplegarlas, pudiendo así acceder a cada uno de los botones que permiten activar/desactivar los respectivos campos booleanos.

- **Listas de cadenas**: Muchas propiedades de distintos componentes permiten almacenar una lista de cadenas de caracteres, siendo su tipo `TStrings`. Dichas propiedades tienen asociado un editor específico, como el mostrado en la parte derecha de la Figura A.19, que se abre mediante el botón asociado a la propiedad. En la citada figura se ha abierto el editor haciendo clic en el botón de la propiedad `SQL`.

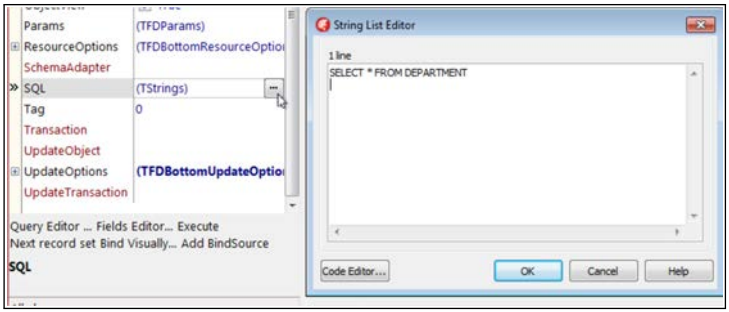


Figura A.19 EDITOR DE LISTAS DE CADENAS DE CARACTERES

- **Objetos**: También es habitual que el tipo de dato correspondiente a una propiedad sea una estructura de datos o un objeto complejo. En estos casos el Inspector de objetos muestra como valor el nombre de tipo y suele ofrecernos dos vías para su manipulación: un botón para desplegar los campos de que está compuesta la propiedad, a la izquierda del nombre, y un segundo botón para abrir un editor específico, a la derecha del valor. Prácticamente hay tantos editores específicos como tipos de datos a medida, ya que cada componente puede registrar sus propios editores. En la Figura A.20 se muestra el editor asociado a la propiedad `Fill` de un formulario, cuyo tipo es `TBrush`.

Los valores asignados a las propiedades utilizando el Inspector de objetos, en la fase de diseño de la interfaz, se almacenarán en el correspondiente archivo `.fmx`/`.dfm`, dependiendo de la biblioteca de componentes que se esté utilizando en el proyecto. Al compilar la aplicación esa información se utilizará para establecer la configuración inicial para cada componente, de forma que al ejecutarse ya contengan los valores seleccionados. Estos podrán modificarse a lo largo de la ejecución del programa desde el código del mismo.

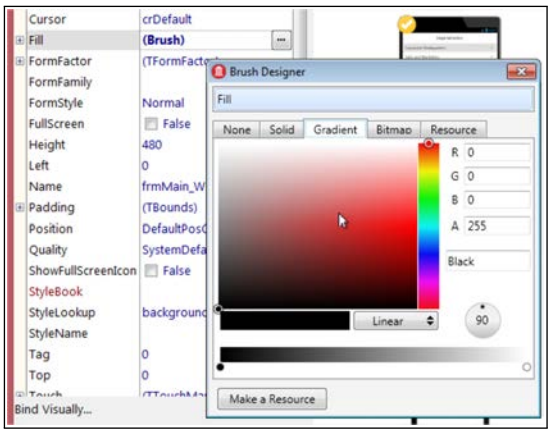


Figura A.20 EDITOR ESPECÍFICO PARA UNA PROPIEDAD DE TIPO `TBRUSH`

A.5 Edición de código

Los formularios, módulos de datos y otros contenedores tienen asociado un m ódulo de código, con extensión `.pas`, en el que se define una clase que representa a dicho objeto, estableciendo cuáles serán los componentes alojados en el mismo, los métodos que implementa, etc. Para manipular el contenido de los módulos de código recurriremos al editor de código. Podemos usar el atajo de teclado **F12** para alternar entre el diseñador de un contenedor y el editor de c ódigo correspondiente.

La mayor parte de los componentes, ya formen parte de la interfaz de usuario o no, son capaces de generar distintos tipos de señales. En ocasiones estas proceden de una acción efectuada por el usuario, por ejemplo al cambiar el contenido de un control. En otras son generadas internamente por el componente para avisar de un cambio de estado, por ejemplo al abrirse o cerrarse una conexión con la base de datos. Estas señales se denominan habitualmente *eventos*.

En esta sección se describe el procedimiento a seguir para gestionar los eventos generados por los componentes, explicándose asimismo varias de las herramientas que el editor nos ofrecer como vías de asistencia a la edición de código, simplificando nuestro trabajo.

A.5.1 Enlace entre eventos y métodos

El Inspector de objetos cuenta, como se ha indicado antes, con una página llamada **EVENTS**. Su finalidad es gestionar los métodos enlazados a los eventos que puede generar cada componente. Cada evento cuenta con dos partes, al igual que las propiedades, siendo la primera un nombre y la segundo un valor. Este siempre será el nombre de un método agregado al módulo de código asociado al contenedor, ya sea formulario, módulos de datos u otro.

Los nombres de los eventos, como se aprecia en la Figura A.21, suelen ser bastante descriptivos, por lo que es fácil determinar en qué momento se ejecutará el método correspondiente. Cada evento solo puede estar conectado con un método, siendo este implementado por la clase que representa al contenedor donde esté el componente[3].

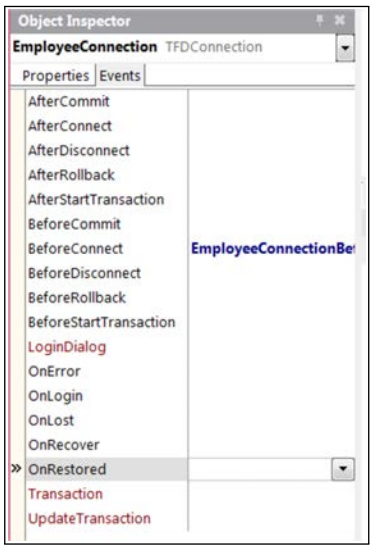


Figura A.21 LISTA DE EVENTOS DE UN COMPONENTE `TFDCONNECTION`

[3]En el módulo habrá definida una clase que derivará de `TForm`, en el caso de los formularios, o de `TDataModule`, si el contenedor es un módulo de datos.

El vínculo entre un evento y el método que se ejecutará cuando se produzca puede establecerse de distintas formas, dependiendo de qué evento sea y de si el método asociado ya existe o no. Nos encontramos con los siguientes escenarios:

- **Evento por defecto**: Cada componente cuenta con un evento por defecto. Este suele ser el utilizado más asiduamente. Para abrir el método asociado a dicho evento basta con hacer doble clic sobre el componente. Por ejemplo, al hacer doble clic sobre un botón se abrirá el método asociado al evento `OnClick`, ya que es el más común, estableciéndose automáticamente el vínculo entre el evento y el método.

- **Nuevo método asociado a evento**: Para generar el método correspondiente a un evento cualquiera, ya sea el evento por defecto o cualquier otro, y establecer el vínculo entre evento y método recurriremos al Inspector de objetos. Localizamos en la página **EVENTS** el evento que interesa y hacemos doble clic sobre el recuadro donde debería aparecer el nombre del método. El resultado de esta acción es idéntica al caso anterior, pero en este caso eligiendo un evento determinado.

- **Método compartido de gestión de evento**: Varios eventos pueden compartir un mismo método[4], de forma que se invoque a este cuando se genere cualquiera de ellos. Para enlazar un evento con un método ya definido anteriormente para otro evento debemos desplegar la lista asociada a su nombre, eligiendo dicho método.

A.5.2 Asistencia a la edición de código

Además de los métodos asociados a eventos, en el módulo de código asociada a cada formulario también encontramos la definición de una clase, la importación de módulos externos y directivas de compilación. Para comprender todos estos elementos necesitamos conocer la sintaxis del lenguaje Delphi, tema al que se dedica el siguiente apéndice. Asimismo deberemos aprender a usar el editor de código integrado en el IDE, ya que este ofrece múltiples funcionalidades de asistencia a la escritura de código. Ese es el tema que nos ocupa aquí.

[4]Esto es cierto siempre que los eventos entreguen exactamente la misma lista de parámetros al método, del mismo tipo y en el mismo orden.

El editor de código de Delphi es algo más que un simple editor de texto. Se trata de una herramienta especializada que va analizando el código a medida que lo escribimos, mostrando cuando es necesario listas de objetos y miembros de estos, detalles sobre los parámetros que necesitan los métodos, etc. Toda esta asistencia proviene de una funcionalidad del IDE denominada *Code Insight*.

Asumiendo que estuviésemos escribiendo el código de un método de gestión de evento, por ejemplo tras haber hecho doble clic sobre un botón para generar el esqueleto del método del evento `OnClick`, la secuencia de acciones que podríamos llevar a cabo con ayuda de Code Insight sería la siguiente:

- Con el atajo **CTRL+BARRA ESPACIADORA** abrimos la lista de objetos accesibles desde el contexto actual. Esta nos ofrece acceso inmediato a las variables, métodos y componentes existentes en el ámbito del método que estamos escribiendo, permitiéndonos elegir (véase la Figura ObjectsList) cualquiera de ellos. Al pulsar **INTRO** el nombre del objeto se introduce en el editor de código.

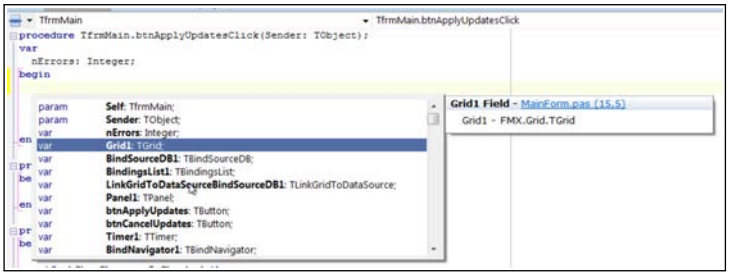


Figura A.22 LISTA DE OBJETOS ACCESIBLES EN EL CONTEXTO ACTUAL

NOTA

Si usamos el citado atajo de teclado tras haber escrito algo en el editor, por ejemplo las iniciales de un objeto, la lista al abrirse mostrará únicamente los elementos cuyo nombre comience por ellas. Si hay una única coincidencia se introducirá automáticamente el nombre completo en el editor.

- Asumiendo que hemos seleccionado de la lista previa uno de los componentes existentes en el formulario, y que queremos acceder a alguno de sus miembros, introduciríamos un punto detrás de su nombre. Esto provocará la apertura inmediata de una lista como la mostrada en la figura A.23. La primera columna de la lista indica la categoría del miembro: propiedad, constante, procedimiento, función, etc. La segunda muestra el nombre del miembro y, según la categoría a que pertenezca, el tipo de dato que le corresponde, la lista de argumentos que acepta, etc. Pulsando **INTRO** se introducirá su nombre en el editor de código.

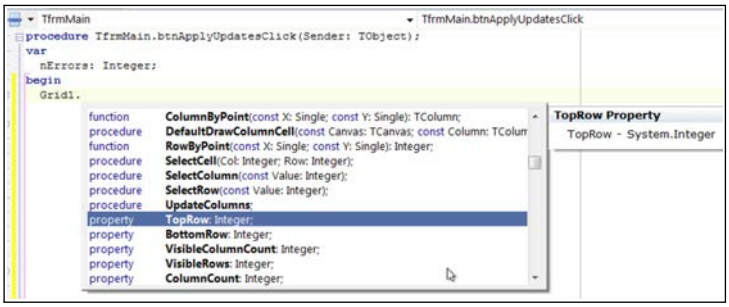


Figura A.23 LISTA DE MIEMBROS DE UN OBJETO

- Al invocar a un método, ya sea procedimiento o función, generalmente tendremos que facilitar unos parámetros de entrada. En cuanto abramos los paréntesis detrás del nombre del método el editor mostrará (véase la Figura A.24) una ventana flotante con el nombre y tipo de cada uno. A medida que vayamos facilitando valores el argumento cuya introducción se espera aparecerá destacado en negrita.

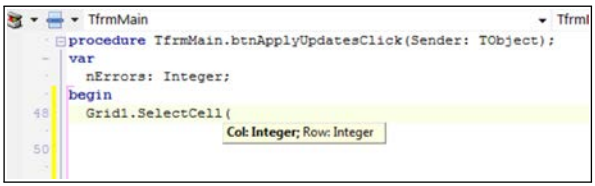


Figura A.24 LISTA DE PARÁMETROS QUE NECESITA EL MÉTODO

- Al facilitar los parámetros de un método o asignar un valor a una propiedad podemos recurrir nuevamente al atajo **CTRL+BARRA ESPACIADORA** para acceder a la lista de objetos disponibles. Al disponer el punto detrás del nombre de un objeto se enumeran todos sus miembros, algo que también afecta a las enumeraciones. Esto facilita, como se aprecia en la Figura A.25, la introducción de valores correctos.

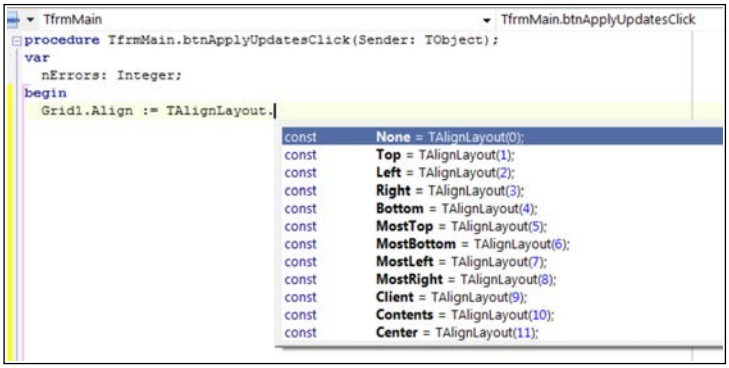


Figura A.25 VALORES VÁLIDOS PARA UNA PROPIEDAD ENUMERADA

Al ir escribiendo código también apreciaremos cómo el editor en ocasiones automatiza ciertas tareas. Al introducir la palabra `begin` y pulsar **INTRO**, por ejemplo, se agregará automáticamente el `end` correspondiente. Esto afecta a otras construcciones comunes del lenguaje Delphi.

NOTA

En el apéndice siguiente, dedicado al lenguaje Delphi, se detallarán algunas herramientas adicionales para la edición de código, como las plantillas y las opciones de refactorización.

A.6 Gestión de plataformas

Los proyectos multi-dispositivo de Delphi pueden ser compilados para múltiples plataformas. Estas aparecerán en el nodo **TARGET PLATFORMS** del proyecto, en el Gestor de proyectos. Concretamente aquí encontramos las plataformas configuradas para el proyecto en cuestión. Existe un subnodo para cada plataforma (véase la Figura A.26), conteniendo a su vez varias ramas más: **TARGET**, **CONFIGURATION** y **LIBRARIES**.

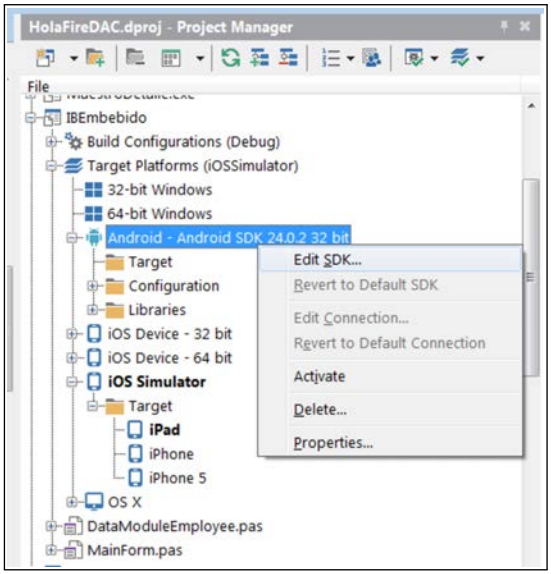


Figura A.26 LISTA DE PLATAFORMAS OBJETIVO Y SU CONFIGURACIÓN

El nodo **TARGET** contendrá los parámetros necesarios para utilizar tanto dispositivos físicos como virtuales, por ejemplo los emuladores de dispositivo iOS y Android. En la carpeta **CONFIGURATION** encontraremos las distintas configuraciones de compilación, por ejemplo con una para depuración y otra para despliegue. Con el menú contextual asociado a cada una de las ramas y sus elementos podremos acceder a las acciones que le afectan.

En cada momento solo una de las plataformas objetivo puede estar seleccionada. Esta será la plataforma para la que se compile el proyecto y en la que se despliegue antes de proceder a su ejecución. En el caso de dispositivos móviles externos, tales como tabletas y teléfonos con iOS y Android, será preciso contar con una conexión por USB y la configuración apropiada[5]. Para cambiar la plataforma objetivo no hay más que hacer doble clic sobre ella. También podemos recurrir a las listas desplegables situadas en la barra de herramientas principal. La primera de ellas nos permite elegir la plataforma y la segunda un dispositivo concreto (véase la Figura A.27). En caso de que el dispositivo deseado no aparezca, el botón que hay a la derecha de la segunda lista permite actualizar el contenido de esta, por ejemplo detectando un dispositivo recién conectado o configurado.

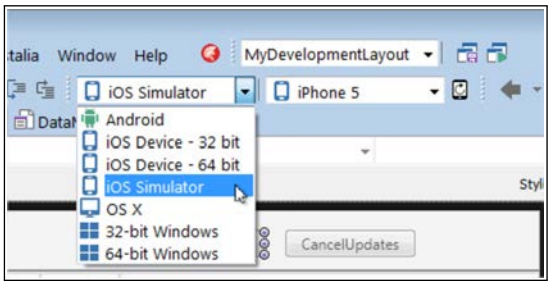


Figura A.27 SELECCIONAMOS LA PLATAFORMA ACTIVA

Desde la versión XE7 de Delphi cada nuevo proyecto de tipo multidispositivo cuenta por defecto con todas las plataformas disponibles, pero en versiones pre-vias esto no es así. Es posible, no obstante, eliminar aquellas plataformas que en principio no interesen. Posteriormente podrían agregarse mediante la opción **ADD PLATFORM** del nodo **TARGET PLATFORMS**, tal y como se muestra en la Figura A.28. El pequeño cuadro de diálogo al que da paso dicha opción cuenta con una lista desplegable que contendrá aquellas plataformas que no existen actualmente en el proyecto, permitiendo agregarlas al mismo.

[5]En el libro *Desarrollo de aplicaciones iOS/Android con Delphi*, de la misma editorial y autor, se describen de forma detallada los procedimientos a seguir para configurar el SDK de Android, XCode sobre OS X, los respectivos emuladores de dispositivos, etc.

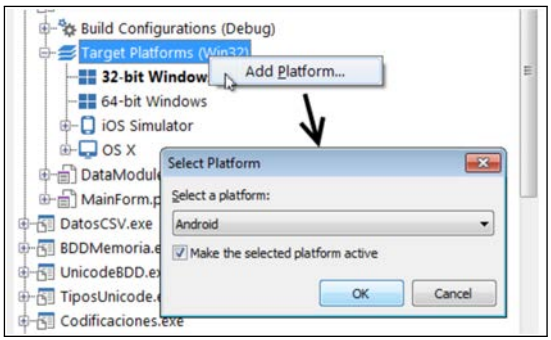


Figura A.28 AGREGAMOS UNA PLATAFORMA OBJETIVO AL PROYECTO

A.7 Generación del proyecto

El fin último del trabajo en un proyecto, insertando y personalizando componentes, diseñando la interfaz de usuario y escribiendo código, es obtener un producto a partir de su generación, generalmente un programa ejecutable para alguna plataforma software. Dicho proceso consta de varias partes, en las que se compila el código para el sistema operativo objetivo, se integran los recursos necesarios y, finalmente, se enlazan todos los elementos obteniendo los módulos a desplegar.

Aunque es posible desplegar y ejecutar el proyecto en un dispositivo iOS/Android y depurar el código de forma remota, con el entorno ejecutándose en un equipo en Windows, en general siempre será más cómodo y rápido hacer este trabajo directamente en el equipo de desarrollo.

A.7.1 Compilación

Podemos lanzar el proceso de compilación del proyecto de múltiples formas, incluyendo la opción **BUILD** que aparece tanto en el menú **PROJECT** (menú principal) como en el menú contextual del proyecto (Gestor de proyectos). También podemos usar el atajo de teclado **MAYÚS+F9** para ejecutar la misma acción. Al finalizar este proceso de construcción, durante el cual se compilarán todos

los módulos de código existentes en el proyecto, se puede obtener una confirmación de que todo ha ido bien o, por el contrario, una lista de los errores que se han encontrado. Haciendo clic en un mensaje accederemos a la línea de código afectada.

Al compilar un proyecto se utilizará la configuración actualmente activa en la rama **BUILD CONFIGURATIONS**, en el Gestor de proyectos. Por defecto siempre encontraremos dos configuraciones predefinidas: **DEBUG** y **RELEASE**. Mediante las opciones del menú contextual es posible, como se aprecia en la Figura A.29, crear nuevas configuraciones, construir el proyecto con la configuración elegida, etc.

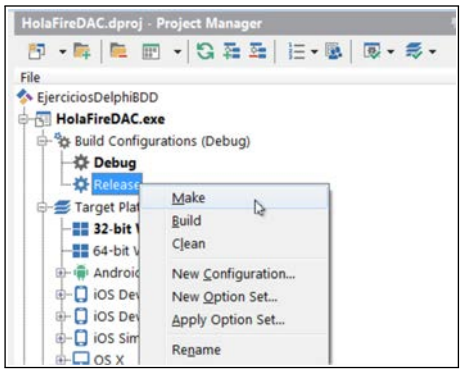


Figura A.29 SELECCIONAMOS LA CONFIGURACIÓN DE GENERACIÓN DEL PROYECTO

La configuración de depuración incluirá en el ejecutable la información necesaria para que el depurador, descrito en el siguiente apartado, pueda realizar su trabajo. Dicha información no será precisa cuando el proyecto vaya a ser puesto finalmente en explotación, por lo que en ese momento se generará con la configuraciób final o **RELEASE.**

A.7.2 Ejecución y depuración

Ejecutar el proyecto desde el propio entorno nos permite comprobar su funcionalidad a medida que avanzamos en el desarrollo, así como depurar el código que hubiésemos escrito a fin de localizar y corregir errores. Encontraremos la

opción **RUN**, asociado al atajo de teclado **F9**, en la barra de herramientas principal, el menú principal y el Gestor de proyectos. Esta opción compilará el proyecto si es necesario, incluyendo la información necesaria para la depuración, procediendo a iniciar la ejecución si no se producen errores.

También podemos usar la opción **RUN WITHOUT DEBUGGING** para ejecutar el proyecto sin incluir información de depuración. Esta configuración sería similar a la usada para desplegar la aplicación y, por ejemplo, permite comprobar cómo reacciona el programa ante excepciones y problemas no controlados por el IDE, como sí ocurre cuando se incluye dicha información.

Asumiendo que hemos lanzado la ejecución del proyecto incluyendo información de depuración, en los siguientes apartados se describen las herramientas de depuración fundamentales que nos ofrece el IDE de Delphi.

Pausar la ejecución del proyecto

En principio el proyecto cuya ejecución hemos lanzado no se detendrá hasta que cerremos la ventana principal del programa o detengamos el módulo de servidor correspondiente, salvo que se produzca algún tipo de excepción que la interrumpa. Podemos, no obstante, detener esa ejecución de distintas formas:

- Mediante la opción **PROGRAM PAUSE** del menú **RUN**, o el botón equivalente de la barra de herramientas principal, la ejecución del programa se detendrá en su punto actual, activándose en el IDE la configuración de depuración.
- Antes de iniciar la ejecución podemos establecer un punto de interrupción en el lugar donde nos interese que se detenga. Para ello podemos usar uno de los siguientes procedimientos:
  - Colocar el cursor de texto en la línea deseada y pulsar la tecla **F5** para activar/desactivar el punto de interrupción. Un punto rojo en el margen izquierdo de la línea indicará que está activo.
  - Hacer clic en el margen izquierdo de la línea, donde aparece el punto antes indicado, para activarlo/desactivarlo.
  - Abrir el menú contextual del editor de código (véase la Figura A.30), abrir el menú **DEBUG** y elegir la opción **TOGGLE BREAKPOINT**.

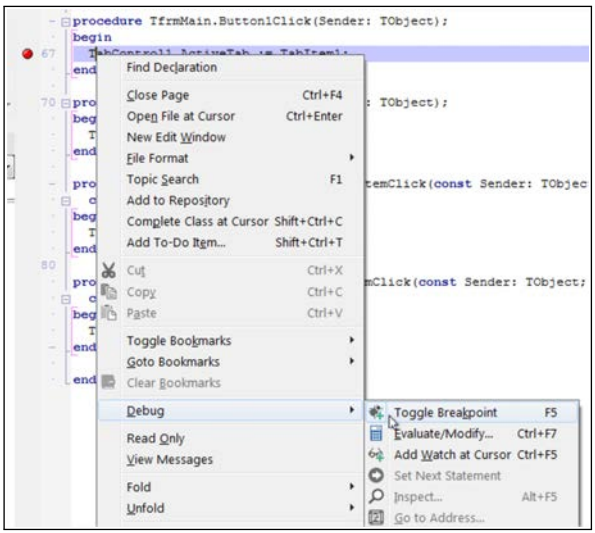


Figura A.30 ACTIVAMOS UN PUNTO DE INTERRUPCIÓN.

- Tras haber iniciado la ejecución del proyecto podemos activar un punto de interrupción en una línea de código por la que sabemos va a pasar el flujo de ejecución, por ejemplo el método del evento `OnClick` de un botón. La gestión del punto de interrupción sería la misma descrita más arriba.

Los puntos de interrupción agregados con los anteriores procedimientos son incondicionales. Esto implica que detendrán la ejecución del programa cada vez que se pase por ese punto, sin que importe el estado de la aplicación. Mediante la opción **RUN—ADD BREAKPOINT** pueden configurarse puntos de interrupción condicionales, mediante los cuales se detendría la ejecución solo si se cumple una determinada condición, una vez que se ha pasado por el punto indicado un número concreto de veces o una combinación de estas casuísticas.

Al seleccionar dicha opción se abrirá un cuadro de diálogo como el de la Figura A.31, en el que podremos introducir la configuración del nuevo punto de interrupción. Al hacer clic en **OK** este aparecerá en el editor como los demás, pero al situar el puntero del ratón sobre el círculo rojo que lo representa obtendremos los parámetros que determinan su funcionamiento.

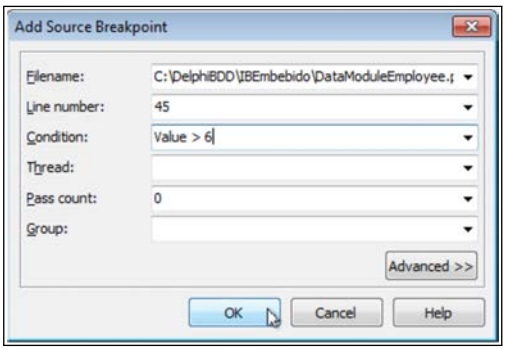


Figura A.31 CONFIGURACIÓN DE PUNTO DE INTERRUPCIÓN

Una vez que la ejecución del proyecto ha sido pausada, con independencia de cuál de los métodos que acaban de describirse se haya utilizado, nos encontraremos en el IDE en su configuración de depuración. La sentencia actual, pendiente aún de ejecutar, aparecerá destacada sobre el resto. En esta pueden llevarse a cabo tareas como la ejecución paso a paso, inspección de objetos y variables, etc. Son las acciones descritas en los siguientes apartados.

Ejecución paso a paso

La ejecución paso a paso consiste en, una vez detenido el programa, ejecutar pequeños bloques de código, tan pequeños que pueden ser sentencias aisladas. Los comandos fundamentales para esta tarea los encontraremos en la barra de herramientas principal, junto a los botones de pausa y parada del programa (véase la Figura A.32).

Estos comandos, que también encontraremos en el menú **RUN**, junto con sus atajos y finalidad, son los siguientes:

- **TRACE INTO**: Ejecuta la sentencia en curso. En caso de que en está haya invocaciones a funciones o procedimientos se pasará a activar la primera sentencia de la primera llamada, permitiendo seguir con la ejecución paso a paso del mismo. El atajo de teclado es **F7**.

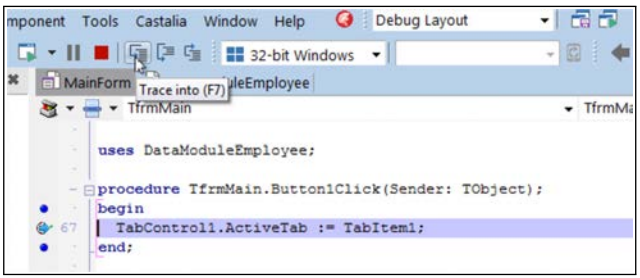


Figura A.32 COMANDOS DE EJECUCIÓN PASO A PASO

- **STEP OVER**: Ejecuta la sentencia en curso, incluyendo todas las llamadas que pudiera implicar esta acción. Es el comando a usar cuando no estamos interesados en depurar paso a paso los métodos a los que se invoca. El atajo de teclado es **F8**.
- **RUN UNTIL RETURN**: Ejecuta desde la sentencia en curso hasta salir de la función o procedimiento a que pertenece. Es el comando adecuado cuando hemos entrado a depurar un método paso a paso pero, llegados a un punto, no nos interesa continuar hasta el final del mismo sino volver al punto de llamada. El atajo de teclado es **MAYÚS+F8**.
- **RUN TO CURSOR**: Ejecuta desde la sentencia en curso hasta aquella en la que hayamos situado el curso en el editor de código. Esto nos permite ejecutar un bloque de sentencias hasta un punto concreto sin necesidad de agregar otro punto de interrupción. El atajo de teclado es **F4**.
- **RUN**: Es el mismo comando que usaríamos para lanzar la ejecución. Durante la sesión de depuración permite continuar con el flujo normal de ejecución de la aplicación hasta alcanzar un punto de interrupción o finalizar. Como sabemos, el atajo de teclado es **F9**.
- **PROGRAM RESET**: La finalidad de este comando es terminar la sesión de depuración, deteniéndola en el punto actual. El atajo de teclado correspondiente es **CTRL+F2**.

A medida que avanzamos en la ejecución, usando los comandos previos, podremos comprobar si el efecto generado por el programa es el esperado, por

ejemplo la información mostrada en la interfaz de usuario. También podremos recurrir a las herramientas explicadas en el siguiente apartado, examinando el contenido de las variables.

Examinar variables y objetos

Una de las verificaciones más usuales durante una sesión de depuración consiste en comprobar si una cierta variable, por ejemplo una propiedad de un objeto usado en la aplicación, contiene el valor que debería. También es habitual cambiar dicho valor para comprobar cómo se comportan las sentencias ejecutadas a continuación.

La vía más inmediata para leer el contenido actual de cualquier variable, siempre que estemos con el programa detenido en una sesión de depuración, consiste en situar el puntero del ratón sobre el identificador a comprobar. Esto provocará, como se aprecia en la Figura A.33, que el contenido de esa variable aparezca en una ventana flotante. Para valores complejos, como objetos o estructuras de datos, esa ventana emergente contará con un botón + que irá dando paso a nuevas ventanas emergentes, tantos niveles como sean precisos, facilitando un examen más profundo.

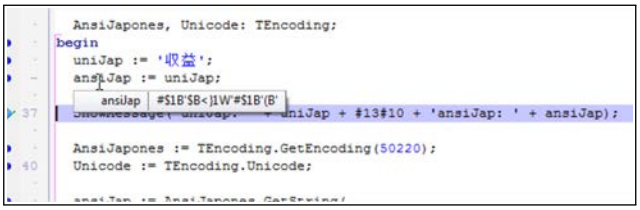


Figura A.33 EL VALOR DE LA VARIABLE APARECE EN UNA VENTANA EMERGENTE

Tanto en el submenú **DEBUG** del menú contextual del editor como en el submenú **RUN** del menú principal encontraremos comandos adicionales que facilitan esta tarea. Son los siguientes:

- **INSPECT**: Tras situar el cursor sobre un identificador, al usar este comando se abrirá una ventana flotante permanente con su nombre, tipo y contenido. Si no hay un identificador bajo el cursor, previamente se pedirá la expresión que se desea evaluar. El atajo de teclado asociado es **ALT+F5**.

- **EVALUATE/MODIFY**: Este comando abre un cuadro de diálogo como el de la Figura A.34. En el apartado **EXPRESSION** se espera una expresión valida, puede contener nombres de objetos con referencias a sus miembros, así como expresiones aritméticas, relacionales y lógicas. El resultado de la expresión se mostrará al hacer clic en el botón EVALUATE. En caso de que la expresión sea modificable, por ejemplo el nombre de una variable o una propiedad, podemos introducir un nuevo valor en la parte inferior y hacer clic en el botón **MODIFY**. Los otros dos botones permiten ejecutar sobre la expresión dada los comandos **INSPECT** y **ADD WATCH**. El atajo de teclado asociado es **CTRL+F7**.

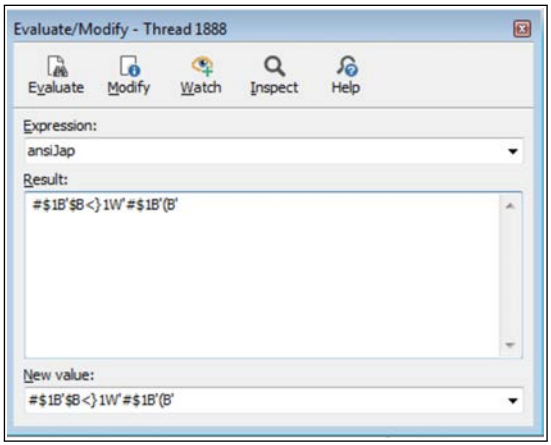


Figura A.34 CUADRO DE DIÁLOGO PARA EVALUACIÓN DE EXPRESIONES

- **ADD WATCH**: Los dos comandos previos facilitan un acceso puntual al contenido de una variable o el resultado de una expresión. Cuando esa información es precisa de forma muy frecuente, a medida que se ejecuta paso a paso el código del proyecto, la selección continua de la misma opción resultará tediosa. En la configuración de depuración del IDE, por defecto en el margen izquierdo, encontramos dos ventanas llamadas **LOCAL VARIABLES** y **WATCH LIST**. La primera muestra automáticamente el contenido de las variables locales al contexto actual. A la segunda podemos agregar otras expresiones a evaluar de forma continua, usando para ello la opción **ADD WATCH** o el atajo de teclado **CTRL+F5**. La expresión a evaluar, así

como las opciones de formato con que se quiere visualizar su contenido, se introducirá en un cuadro de diálogo como el mostrado en la Figura A.35.

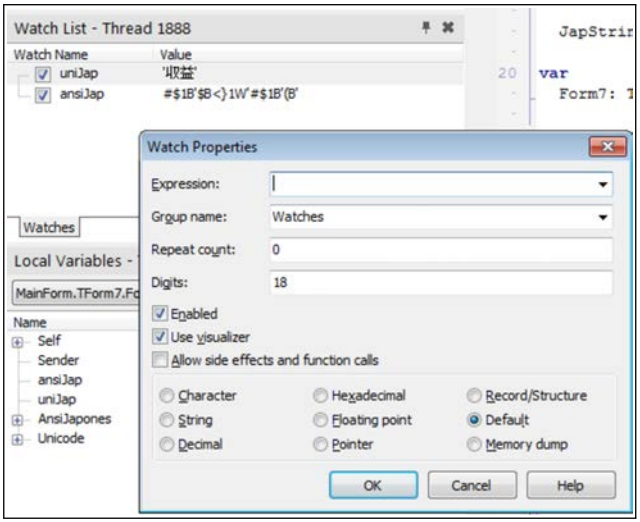


Figura A.35 VENTANA CON VARIABLES BAJO SUPERVISIÓN

La pila de llamadas

Al ejecutar paso a paso, con los comandos antes enumerados, somos conscientes de la línea que ha seguido el flujo de ejecución hasta alcanzar la sentencia en curso. Esto no es así, sin embargo, cuando se detiene inicialmente la ejecución, por ejemplo mediante un punto de interrupción. A un cierto método puede llegarse desde distintos puntos de la aplicación, información que puede ser esencial en algunos casos.

Otro de los paneles abiertos por defecto en la configuración de depuración, inicialmente situado en la parte superior izquierda del IDE, es **CALL STACK**. Este siempre muestra la pila de llamadas que ha llevado la ejecución hasta la sentencia en curso, indicando el nombre del objeto, el método y una dirección. De esta forma es fácil saber el camino que ha seguido el flujo de ejecución. Como puede comprobarse en la Figura A.36, el menú contextual de dicho panel

permite acceder al código fuente de la línea seleccionada, así como comprobar las variables locales asociadas.

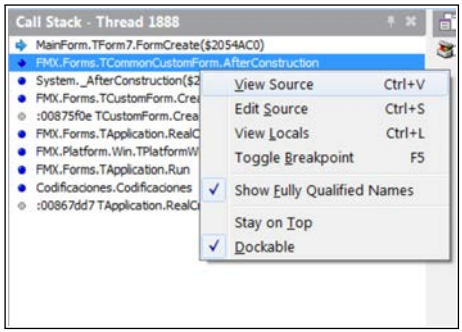


Figura A.36 INFORMACIÓN SOBRE LAS LLAMADAS HASTA EL PUNTO ACTUAL

> **NOTA**
>
> *Si cualquiera de los paneles mencionados en los apartados previos no estuviese visible, podemos mostrarlo con la opción adecuada del menú* **VIEW—DEBUG WINDOWS**.

Además de las ya descritas, el entorno de Delphi cuenta con herramientas adicionales de depuración, como las que examinan el código ensamblador correspondiente a las sentencias o el estado de la CPU y la FPU. Raramente necesitaremos utilizarlas salvo que trabajemos a bajo nivel.

A.8 Resumen

Este apéndice ha ofrecido una vista general del entorno de desarrollo de Delphi, describiendo muchas de las herramientas con que cuenta y cuya finalidad es facilitar la gestión de proyectos, diseño de interfaces, edición de código y depuración. A medida que usemos Delphi, poniendo en práctica los ejercicios

propuestos en los capítulos de este libro, la experiencia nos permitirá sacar el mayor provecho de estas herramientas.

Además de las enumeradas en este apéndice, el IDE de Delphi cuenta con opciones y utilidades adicionales que no han sido mencionadas. Muchas de ellas serán descritas con detalle en los capítulos de este libro. Un ejemplo de ello es el Explorador de datos, el diseñador de enlaces entre datos y controles o el control de versiones con Git.

APÉNDICE B
EL LENGUAJE DELPHI

En los distintos capítulos de este libro tendremos que escribir habitualmente código, principalmente como parte de los proyectos propuestos a modo de ejercicios. La mayoría de las veces ese código son apenas unas cuantas sentencias, siempre asociadas a métodos cuyo nombre y lista de parámetros ya están predefinidos al estar vinculados a eventos. Además muchas de esas sentencias son simples asignaciones de valores a algunas propiedades o bien llamadas a otros métodos.

En el contexto de dichos ejercicios es más importante conocer la finalidad de cada propiedad y método que entrar en los detalles relativos a cómo se implementaba cada funcionalidad. Por dicha razón no se entra en formalizar muchos detalles relativos al lenguaje Delphi.

En este apéndice van a introducirse las estructuras más relevantes del lenguaje de programación Delphi. No obstante, se asume que el lector tiene unas nociones básicas y generales sobre programación, conociendo qué es una variable, qué es una expresión o cómo funciona un condicional. Estos son aspectos no abordados aquí, como tampoco lo son las bases de la programación orientada a objetos. Nos concentraremos exclusivamente en cómo dichos elementos toman forma en el lenguaje Delphi. En consecuencia, si la programación es algo nuevo para ti es recomendable que recurras a un libro de introducción, más básico que este.

B.1 Sintaxis básica

La sintaxis de Delphi es distinta a la de los lenguajes de programación más usados actualmente, como pueden ser Java, C++, C# y JavaScript, ya que dicha sintaxis no está derivada de C sino de Pascal. Este hecho conlleva las siguientes implicaciones:

- En general el lenguaje no diferencia entre mayúsculas y minúsculas. Tanto las palabras clave como los identificadores creados por el usuario, por ejemplo nombres de variables y métodos, pueden ser escritos usando distintas combinaciones de mayúsculas y minúsculas y, a pesar de todo, el compilador los seguirá reconociendo sin problemas.
- Los bloques de código se delimitan mediante palabras clave del lenguaje, concretamente `begin` y `end`, en lugar de utilizar llaves o algún otro tipo de símbolo.
- A la hora de declarar variables, el tipo de las mismas se indica tras el nombre, no antes, y se utiliza un carácter de dos puntos como separación entre la lista de nombres de variables y el tipo.
- A diferencia de lo que ocurre con la mayoría de los derivados de C, en general Delphi no precisa el uso de paréntesis para delimitar expresiones asociadas a algunas instrucciones.

- Los operadores lógicos tienen nombres, como pueden ser `or`, `and` y `not`, en lugar de estar representados por símbolos del tipo `||`, `&&` y `!`.

El estilo sintáctico puede agradar a unos y no a otros, dependiendo de nuestras preferencias personales. No obstante, en términos generales el estilo del código en Pascal tiende a ser más fácil de leer que el código al estilo de C. Sin embargo, este último produce normalmente código más compacto que el primero, ya que muchas de las palabras clave y operadores son sustituidas por símbolos.

Vamos a comenzar, en la sección siguiente, analizando la estructura general de los módulos de código escritos con Delphi. Una vez que tengamos una visión general de cuáles son las secciones existentes, estudiaremos varios temas con algo más de profundidad. Entre otros se tratarán los tipos de datos con que cuenta el lenguaje, las estructuras de control básicas y cómo abordar la definición de una clase.

B.1.1 Módulos de código Delphi

Cada vez que creemos un nuevo módulo de código Delphi podremos encontrarnos básicamente con una de dos posibles estructuras, dependiendo de si se trata de un módulo de programa o un módulo estándar (también conocido como *unit*).

El programa es el único módulo de código por el que puede iniciarse la ejecución de una aplicación. Por esta razón únicamente puede existir un módulo de dicho tipo en cada proyecto. Por el contrario, los módulos estándar son meros contenedores de código, utilizados para alojar definiciones de clases, definiciones de estructuras de datos, implementación de funciones, etc. Por tanto en un proyecto puede haber tantos módulos de tipo *unit* como sea preciso. Según los casos, dichos módulos pueden estar vinculados a formularios.

En un proyecto de aplicación móvil Delphi típico, como los desarrollados a modo de ejercicio en las dos primeras partes de este libro, habrá un módulo de código independiente asociado a cada uno de los formularios. El nombre de cada módulo será el mismo dado al formulario al que pertenece, pero añadiendo la extensión `.pas`. Asimismo, también los módulos de datos tendrán asociado un módulo estándar de código.

El proyecto propiamente dicho también está vinculado a un archivo, con extensión `.dpr`, en el que se encuentra almacenado el código encargado de poner

en marcha la aplicación. Puedes ver este código en el editor de Delphi abriendo el menú contextual del proyecto (en el Gestor de proyectos) y eligiendo la opción **VIEW SOURCE**. Analizamos la estructura de dicho módulo en la siguiente sección.

B.1.2 Estructura del módulo de programa

Si abrimos el módulo de código asociado a cualquiera de los proyectos desarrollados en este libro, encontraremos un código prácticamente idéntico al mostrado a continuación:

```
1  program MaestroDetalle;
2
3  uses
4    System.StartUpCopy,
5    FMX.Forms,
6    MainForm in 'MainForm.pas' {frmMain},
7    DataModule in 'DataModule.pas' {dmMaestroDetalle: ↙
         ↳ TDataModule};
8
9  {$R *.res}
10
11 begin
12   Application.Initialize;
13   Application.CreateForm(TfrmMain, frmMain);
14   Application.CreateForm(TdmMaestroDetalle, dmMaestroDetalle↙
         ↳ );
15   Application.Run;
16 end.
```

Listado B.1 Código del módulo `.dpr` de un proyecto

En Delphi un módulo de código que actúa como punto de entrada a una aplicación siempre comienza con una cabecera en la que aparece la palabra clave `program`. Esta irá seguida del nombre del programa (el proyecto). El bloque de sentencias delimitado por las palabras clave `begin` y `end` es el punto por el que comenzará la ejecución.

Entre la cabecera del módulo y el citado bloque de sentencias ejecutables pueden aparecer varias secciones. La más común es la que se inicia con la palabra clave `uses`, enumerando las referencia a módulos externos que son necesarios para el funcionamiento del programa. Los nombres de módulos en la lista irán separados por comas, usándose un punto y coma como indicador de fin.

Si nuestro proyecto contase con más de un formulario, siendo uno de ellos responsable de mostrar el otro, tendríamos que introducir en el módulo de código del primero una referencia al segundo. Esta operación puede completarse por dos vías distintas:

- Añadiendo manualmente una sección `uses` en el primer módulo, incluyendo el nombre del módulo en el que se define el segundo formulario.
- Utilizando la opción **FILE—USE UNIT** del menú principal. Esta abre un pequeño cuadro de diálogo con una lista en la que aparecen todos los formularios del proyecto, pudiendo elegir el que nos interese.

Esta misma técnica sería la que se utilizaría para hacer referencia a un módulo de datos desde un formulario, a fin de poder enlazar las información procedente del primero con la interfaz diseñada en el segundo.

Otra de las secciones opcionales que podemos encontrar en un módulo de programa, antes del bloque de sentencias a ejecutar, es la sección de declaración de variables. Esta se inicia con la palabra reservada `var`, tras la cual se facilitará una lista de nombres de variables y tipos de datos. Es posible facilitar varias variables separando sus nombres mediante comas. Al final de la lista ha de indicarse el tipo de dato al que pertenecerían, separando lista y tipo por un carácter dos puntos. Cada bloque de declaración se finalizaría con un punto y coma.

Un detalle importante que debemos tener en cuenta es que, en general, todas las sentencias en Delphi finalizan con un punto y coma, pero tras la palabra reservada `end` que denota el fin del módulo de programa se usa un punto en su lugar.

NOTA

Cuando se está acostumbrado a la sintaxis de lenguajes como C, C++ o Java, suele ser habitual que automáticamente añadamos un punto y coma...

...al final de todas las sentencias, algo que en Delphi generará errores de compilación en algunas situaciones como tendremos ocasión de ver más adelante.

B.1.3 Estructura de un módulo de código estándar

Los módulos de código en Pascal, y por tanto en Delphi, están pensados para alojar código que en algún momento será utilizado desde el módulo principal (el programa) o bien desde otros módulos. Por esta razón la estructura de dichos módulos no es la misma descrita en el punto anterior.

Todos los módulos estándar Delphi están estructurados en dos secciones: interfaz e implementación[1]. La cabecera, conteniendo el nombre del módulo, se inicia con la palabra reservada `unit` en lugar de `program`, como puede apreciarse en la siguiente plantilla de código:

```
1  unit MyUnitName;
2
3  interface
4
5  uses unitName1, unitName2, ..., unitNameN;
6
7  type
8
9   Definiciones de tipos de datos
10
11 var
12
13  Declaración de variables
14
```

[1]Opcionalmente es posible añadir dos secciones más a un módulo estándar, pensadas para contener el código de inicialización y el de finalización. Dichas secciones se iniciarían con las palabras clave `initialization` y `finalization`, respectivamente.

```
15 implementation
16
17  Métodos y tipos de uso interno
18
19 end.
```

Listado B.2 Plantilla con la estructura de un módulo estándar

La sección `interface` establece qué elementos definidos en el módulo en curso estarán disponibles para otros módulos. Habitualmente en esta sección se definen tipos de datos y se declaran variables.

Si, por ejemplo, examinamos el módulo asociado al formulario de uno de nuestros proyectos, encontraremos en la sección `interface` la definición de la clase `TForm1`, así como la declaración de una variable llamada `Form1` de tipo `TForm1`. Esto permite a otros módulos utilizar ese formulario, accediendo a sus propiedades y llamando a los métodos que pudiera ofrecer.

En cuanto a la sección `implementation`, puede contener definiciones de tipos de datos, declaraciones de variables y, sobre todo, el código de implementación correspondiente a los elementos previamente definidos en la sección de interfaz. Una vez que el módulo se haya compilado, ninguno de los detalles de la sección `implementation` será visible desde el exterior, al contrario de lo que ocurre con el de la sección `interface`, del que podría obtenerse información general gracias a un mecanismo conocido como RTTI (*Run-Time Type Information*).

B.1.4 Referencias a módulos

Habitualmente la primera parte de cualquiera de las dos secciones existentes en un módulo de código estándar se inicia con la palabra reservada `uses`, dando paso a la lista de referencias a otros módulos en cuya sección `interface` habrá definidos tipos de datos, variables o métodos que pretendemos usar desde el módulo actual.

Uno de los proyectos de ejemplo[2] que se distribuyen con Delphi, llamado `Forms`, cuenta con dos formularios llamados `PortraitForm` y `LandscapeForm` (véase la Figura B.1), pensados para su uso con una orientación de pantalla específica.

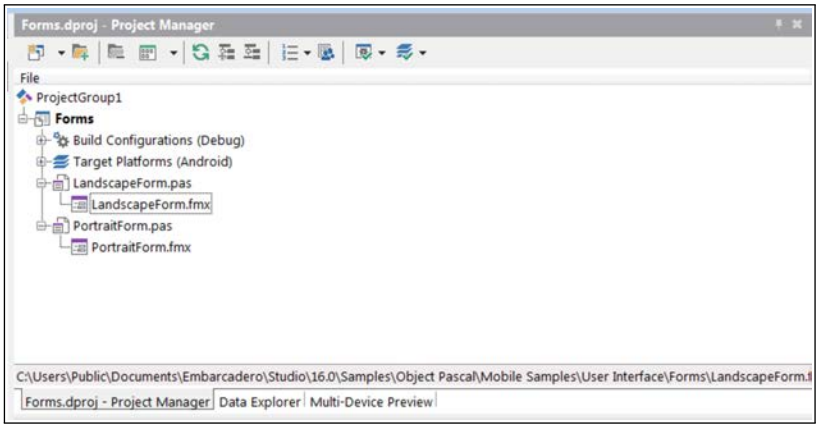


Figura B.1 FORMULARIOS EN EL PROYECTO DE EJEMPLO `FORMS`

Abriendo el módulo de código de cualquiera de ellos, podremos ver que las primeras sentencias de su sección `implementation` son las siguientes:

```
1 {PortraitForm.pas}
2 ...
3 implementation
4
5 uses
6   LandscapeForm;
7 ...
8 {LandscapeForm.pas}
```

[2]Los proyectos de ejemplo incluidos con el producto se encontrarán normalmente en la carpeta de documentos disponible para todos los usuarios. La barra de estado de la ventana mostrada en la Figura B.1, en la parte inferior, indica la ruta completa donde debería estar almacenado el proyecto.

```
9  ...
10 implementation
11
12 uses
13   PortraitForm;
14 ...
```

Listado B.3 Referencias entre formularios

De esta forma el código del módulo `PortraitForm.pas` puede mostrar el formulario `LandscapeForm` cuando se detecta un cambio en la orientación de la pantalla, siendo el comportamiento del módulo `LandscapeForm.pas` similar al anterior.

Los componentes FMX, al igual que los servicios básicos ofrecidos por la RTL[3], están distribuidos entre un conjunto mucho mayor de módulos. Como regla general, no necesitaremos añadir manualmente referencias a dichos módulos en nuestro código, en la cláusula `uses` de nuestros módulos, dado que el diseñador de Delphi se ocupa de ello por nosotros.

No obstante hay excepciones, como cuando necesitamos usar servicios que no están directamente relacionados con componentes agregados a los formularios. Para acceder a ciertos servicios depdendientes de la plataforma, por ejemplo los servicios de eventos de aplicación, es necesario incluir una referencia al módulo `FMX.Platform`.

Teniendo en cuenta que aparte de la RTL Delphi incluye dos bibliotecas de componentes más, la FMX y la VCL (*Visual Component Library*), cada una de las cuales aporta un gran conjunto de módulos, no es raro que a veces encontremos servicios muy parecidos en varias de ellas, así como módulos que tienen el mismo nombre. Tanto la RTL como la FMX cuentan con un módulo `Types` en el que se alojan definiciones básicas de tipos. Análogamente, tanto en la FMX como en la VCL existe un módulo llamado `Controls`, en el que se definen controles de interfaz de usuario.

A fin de evitar colisiones de identificadores y potenciales ambigüedades, para referenciar cualquier módulo se utilizan referencias completas. Esta es la razón

[3]Los módulos de la FMX usan el prefijo `FMX`, mientras que los módulos pertenecientes a la RTL normalmente usan el prefijo `System`.

de que se utilicen prefijos dependientes de la biblioteca delante de los nombres de los módulos, por ejemplo: `System.Types` y `FMX.Types`, `Vcl.Controls` y `FMX.Controls`, etc.

Esta denominación compuesta se establece cada vez que se crea un nuevo módulo de código. En ese momento, el nombre completo ha de ser indicado tras la palabra reservada `unit`. El nombre del archivo en que se guardará el módulo será el mismo, añadiendo la extensión `.pas`.

B.1.5 Comentarios

Al igual que otros muchos lenguajes de programación, Delphi contempla la inclusión tanto de comentarios que finalizan al final de la línea actual como de bloques de comentarios. Los primeros se inician con los caracteres `//` y se extienden hasta el final de la línea en curso. Todo el texto que aparezca tras `//` será ignorado completamente por el compilador. El siguiente sería un ejemplo de uso de este tipo de comentario:

```
1 ...
2 if EventInfo.GestureID = igiZoom then
3 begin
4   if iLastDistance > EventInfo.Distance then // Zoom out
5     Label1.Font.Size := Label1.Font.Size - 1
6   else // Zoom in
7     Label1.Font.Size := Label1.Font.Size + 1;
8 ...
```

Listado B.4 Comentarios hasta fin de línea

Los bloques de comentarios se delimitan con el carácter `{` al inicio y el carácter `}` al final. También se permite el uso de las parejas de caracteres `(*` y `*)` en lugar de los anteriores. Este tipo de comentario es útil para documentar los parámetros que se esperan en un método, por poner un ejemplo, junto con otros detalles como el tipo del valor devuelto o las precondiciones que sean aplicables.

Delphi trata de manera especial aquellos comentarios que comienzan con la palabra `TODO` seguida de dos puntos y un texto. Por ejemplo:

```
1 procedure TForm1.FormCreate(Sender: TObject);
2 begin
3   { TODO: Restoring application state to be implemented }
4 ...
```

Listado B.5 Comentario de tipo `TODO`

El texto asociado a estos comentarios aparecerá en una ventana que, por defecto, está oculta. Para hacerla visible usaremos la opción **VIEW—TO-DO LIST**. Como se aprecia en la Figura B.2, hay dos columnas pensadas para mostrar el propietario de la tarea pendiente de realizar y también la categoría de esta. Esos detalles pueden facilitarse usando las opciones `-o` y `-c` tras la palabra `TODO`.

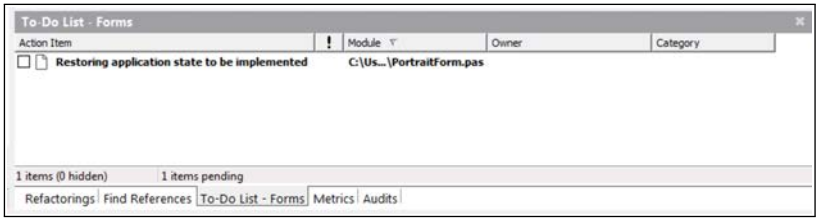


Figura B.2 SEGUIMIENTO AUTOMÁTICO DE LOS COMENTARIOS TIPO `TODO`

B.2 Tipos de datos fundamentales

Una de las partes que pueden existir en un módulo de código, en cualquiera de las secciones del módulo (`interface` o `implementation`) así como dentro de los métodos, es el apartado de declaración de variables. Este se inicia con la palabra clave `var`, tras la cual se dispondrán tantas declaraciones como se precise. Cada declaración se ajustará a la sintaxis mostrada a continuación:

```
1
2 var
3   variable1, ..., variableN: dataType;
4 ...
```

Listado B.6 Sintaxis para declarar variables

Todas las variables indicadas en la lista (delante de los dos puntos), cuyos nombres se separarán con comas, compartirán un mismo tipo de dato, especificado al final de la declaración. Como muchos otros lenguajes de programación, Delphi cuenta con un conjunto básico de tipos de datos para operar con números, caracteres, enumeraciones y cadenas de caracteres, así como tipos compuestos como pueden ser vectores y matrices (*arrays*), conjuntos y registros o estructuras.

Algunos de los tipos numéricos de Delphi cambian su tamaño, y por tanto el rango de valores que pueden tomar, dependiendo de la plataforma para la que se compile el proyecto. Hay dos posibilidades actualmente: plataformas de 32 bits y plataformas de 64 bits. Los caracteres y cadenas de caracteres pueden almacenarse utilizando codificación ASCII o codificación UNICODE (véase el Capítulo 8). La segunda es la que se utiliza por defecto.

En las secciones siguientes se enumerarán los tipos de datos fundamentales de Delphi. Se asume que el lector sabe diferenciar entre tipos de datos numéricos y caracteres, así como que tiene conceptos básicos sobre qué es un vector/matriz o una estructura de datos.

B.2.1 Números enteros y de coma flotante

Los tipos de datos pensados para operar con números son parte del conjunto de tipos de datos básicos de Delphi, estando dividido en dos categorías: tipos enteros y tipos reales. Una variable de tipo entero puede contener cualquier valor de un subconjunto de $\mathbb{Z}$, mientras que una de tipo real, utilizando formato de coma flotante, permite trabajar con valores de un subconjunto de $\mathbb{R}$[4]. El tipo específico que elijamos determinará el tamaño de dichos subconjuntos.

[4]En este contexto $\mathbb{Z}$ y $\mathbb{R}$ han de interpretarse matemáticamente. $\mathbb{Z}$ hace referencia al conjunto de los números enteros y $\mathbb{R}$ representa el conjunto de todos los números reales.

Para elegir un tipo entero comenzaremos decidiendo si necesitamos almacenar el signo o no. ¿Operaremos con números negativos? Además también es importante el rango de valores que sea necesario almacenar. Dependiendo de estos factores, seleccionaremos uno entre la docena de tipos enteros mostrados en la tabla siguiente:

Tamaño	Tipos sin signo	Tipo con signo
8 bits	`Byte`	`ShortInt`
16 bits	`Word`	`SmallInt`
32 bits	`LongWord`	`LongInt`
64 bits	`UInt64`	`Int64`
32 bits	`Cardinal`	`Integer`

Tabla B.1 Tipos de datos enteros

Los tipos `LongInt` y `LongWord` son dependientes de la plataforma a partir de Delphi XE8, siendo su tamaño de 32 bits en todas a ellas a excepción de iOS de 64 bits, en la que son enteros de 64 bits. Asimismo se han agregado dos nuevos tipos en dicha versión del lenguaje: `FixedInt` y `FixedUInt`. Ambos tienen un tamaño fijo de 32 bits sin que importe la plataforma, siendo el primero un tipo con signo y el segundo sin signo.

Además de los anteriores, que se caracterizan por ser tipos de datos que, en general, mantienen su tamaño sin que importe la plataforma para la que se compila el proyecto, también podemos utilizar los tipos `NativeInt` y `NativeUInt`. Estos resultan útiles cuando queremos tener 32 bits o 64 bits dependiendo de cuál sea la plataforma de destino.

En cuanto a los tipos de datos para números reales, tenemos básicamente tres alternativas: `Single`, `Double` y `Extended`. Elegiremos uno de ellos dependiendo de que necesitemos almacenar números en simple precisión, en doble precisión o con precisión extendida. El tamaño de estos tipos de datos es 4, 8 y 10 bytes, respectivamente, y siempre almacenan números con signo.

> **NOTA**
>
> *A pesar de que `Extended` es un tipo numérico en coma flotante de alta precisión, no es aconsejable su uso cuando se desarrollan aplicaciones multi-plataforma ya que tiene menor precisión en Win64 que en otros sistemas como Win32. Los tipos `Single` y `Double` con compatibles entre plataformas.*

Por último, Delphi nos ofrece también un tipo numérico de coma fija, en lugar de flotante, llamado `Currency`. Su tamaño es de 64 bits, encontrándose a medio camino entre los tipos de datos enteros y los de coma flotante. `Currency` es un tipo ideal cuando se necesita trabajar con números que tienen parte fraccionaria, pero sin pérdida de dígitos significativos como ocurre con los tipos `Single`, `Double` y `Extended`.

B.2.2 Caracteres y cadenas de caracteres

Delphi cuenta con tres tipos de datos para operar con caracteres, así como otros tres para trabajar con cadenas de caracteres. En realidad, uno de ellos es un alias para otro tipo, por lo que existen dos tipos distintos. La diferencia entre estos tipos estriba en el mecanismo usado para codificar los caracteres a la hora de almacenarlos, que puede ser ASCII o UNICODE.

> **NOTA**
>
> *En el Capítulo 8 se detalla cómo funcionan las distintas codificaciones de caracteres, incluyendo UTF-8, y el trabajo con cadenas de caracteres UNICODE.*

Los tipos `AnsiChar` y `WideChar` están diseñados para almacenar un carácter ASCII (8 bits) o un carácter UNICODE (16 bits[5]), respectivamente. El tipo

[5]En realidad la longitud de un carácter codificado como UNICODE es variable, pudiendo llegar a los 32 bits si fuese necesario.

`Char` es un sinónimo para `WideChar`, haciendo que UNICODE sea la codificación por defecto usada en Delphi. Por tanto no estamos limitados al reducido conjunto de caracteres occidental, siendo posible usar caracteres de muchos otros alfabetos.

Cuando lo que se necesita es trabajar con cadenas de caracteres[6], Delphi nos ofrece tres tipos de datos posibles: `AnsiString`, `UnicodeString` y `String`. La última es un alias para `UnicodeString`. Como es fácil deducir, la primera almacena cadenas de caracteres con codificación ASCII, mientras que la segunda usa codificación UNICODE.

Por compatibilidad con versiones previas de Delphi en Windows, también se mantiene el tipo de dato `WideString`. Este está pensado para almacenar cadenas de caracteres UNICODE, pero con un formato específico pensado para hacer posible la interoperabilidad con los componentes COM de Windows.

La RTL de Delphi cuenta con un módulo llamado `System.Character` en el que se ofrecen múltiples funciones de utilidad, así como una clase llamada `TCharacter`. Con ellos es posible comprobar la categoría de cualquier carácter, para saber si es un signo de puntuación, un dígito numérico, una letra mayúsculas o minúscula, etc. Además, también hay funciones para convertir códigos numéricos en caracteres y viceversa, convertir letras mayúsculas o minúsculas y similares.

B.2.3 Otros tipos de datos básicos

Aparte de los valores numéricos, los caracteres y las cadenas de caracteres, también pueden considerarse tipos básicos los valores lógicos (o *booleanos*) y los punteros.

El tipo de dato `Boolean` puede contener uno de dos valores predefinidos: `True` y `False`. Estos valores pueden ser obtenidos como resultado de la evaluación de expresiones relacionales, así como ser usados en expresiones lógicas.

[6]En Pascal, y por tanto también en Delphi, las cadenas de caracteres se almacenan en memoria con un prefijo en el que se indica la longitud actual de la cadena. A diferencia de lenguajes como C, no es necesario agregar un carácter `NULL` al final para indicar dónde termina la cadena.

A pesar de que existen otros tipos booleanos, como `ByteBool`, `WordBool` y `LongBool`, su uso no es muy frecuente.

En cuanto a los punteros, su tamaño es siempre coherente con la plataforma para la que se compila el proyecto. Una aplicación que se despliega en una plataforma móvil de 32 bits usará punteros de 32 bits, mientras que en una plataforma de 64 bits, como puede ser Win64, contará con punteros de 64 bits.

Un puntero es una dirección de memoria, apuntando al lugar en el que está almacenado cualquier tipo de dato: números, caracteres, registros, etc. Para declarar un puntero con un tipo específico de dato es necesario usar como prefijo del tipo base el símbolo `^`. Por ejemplo:

```
1 ...
2 var
3   aNumber: Integer;
4   pointToANumber: ^Integer;
5
6 begin
7   pointToANumber := @aNumber;
8   pointToANumber^ := 6;
9 ...
```

Listado B.7 Uso básico de un puntero

En este ejemplo la variable `pointToANumber` es un puntero a un valor de tipo `Integer`. La dirección en la que la variable `aNumber` almacena su valor es obtenida mediante el operador `@` y es almacenada en el puntero. A continuación, usando de nuevo el operador `^`, podemos acceder al contenido al que apunta el puntero a fin de leer o escribir dicho valor, tal y como se hace en la última sentencia.

B.2.4 Notación para literales e identificadores

Para declarar una nueva variable necesitaremos conocer los tipos de datos existentes, algunos de ellos ya han sido introducidos en las secciones anteriores, así como estar habituados a las reglas que han de seguirse para dar nombres a los identificadores. Además, la introducción de valores literales en el código ha de

adecuarse a una notación concreta. Esta notación es específica de cada lenguaje de programación.

Delphi permite a los programadores definir sus propios identificadores, por ejemplo a la hora de dar nombre a variables, métodos o tipos de datos definidos por el usuario, dándoles nombres que pueden tener cualquier longitud. No obstante, el compilador solamente usará los primeros 255 caracteres del identificador. En la práctica, esto es más que suficiente para evitar cualquier conflicto que pudiera darse al duplicarse un identificador.

El primer carácter de un identificador ha de ser una letra o el carácter _. Este irá seguido de una secuencia de letras (mayúsculas o minúsculas, dado que el lenguaje no distingue entre ellas), números y el propio carácter _. Si bien los espacios y símbolos, como pueden ser los operadores, no están permitidos, sí que se contempla el uso de cualquier carácter alfabético UNICODE. Esto significa que podemos usar letras específicas de idiomas ajenos al inglés, como la ñ o letras acentuadas, a la hora de definir identificadores sin ningún problema.

Una regla básica en Delphi, y que es aplicable a casi todos los lenguajes de programación, es que no se permite el uso de palabras reservadas como identificadores. Esto incluye palabras clave del lenguaje, instrucciones, operadores, etc.

En relación a la notación que ha de utilizarse al introducir valores literales en el código, las reglas generales a seguir son las que se indican a continuación:

- Los valores numéricos se introducen tal cual, como una sucesión de dígitos y opcionalmente un punto decimal para separar la parte entera de la fraccionaria. También puede incluir un signo al inicio, de primer carácter.
- Para trabajar con números muy grandes (o muy pequeños) podemos recurrir a la notación exponencial. Tras la secuencia de dígitos que actúa como multiplicador, debe utilizarse el carácter `E` seguido de un exponente que se aplicará a la base 10. Por ejemplo, `45E6` sería equivalente a `45000000`.
- Se permite la introducción de valores numéricos en base hexadecimal utilizando el prefijo `$`. Por ejemplo, `$B800`.
- Los valores de tipo carácter se delimitan entre comillas simples. Por ejemplo, `'b'`. También es posible introducir códigos de carácter, si los conocemos, mediante el prefijo #. Por ejemplo, `#98` sería `'b'`.

- Las cadenas de caracteres también se delimitan con comillas simples. A pesar de no ser algo muy habitual, también es posible introducir secuencias de códigos de carácter[7], con cada código precedido del símbolo `#`. Por ejemplo, `#68#101#108#112#104#105` es lo mismo que `'Delphi'`.

Los valores literales pueden ser asignados a variables, pero también ser utilizados como parámetros a enviar a los métodos, así como valores constantes en expresiones de cualquier tipo.

B.2.5 Enumeraciones

Durante el desarrollo de algunos de los ejercicios propuestos en los capítulos del libro se trabaja con propiedades que solamente pueden tomar un conjunto limitado de valores predefinidos. Estos son conocidos como tipos enumerados. Las propiedades `Align` y `TabPosition` están entre ellas. El tipo de la primera es `TAlignLayout`, mientras que la segunda es de tipo `TTabPosition`.

El tipo `TAlignLayout` está definido en el módulo `FMX.Types` como se muestra a continuación:

```
1 ...
2 type
3   TAlignLayout = (None, Top, Left, Right, Bottom, MostTop,
4       MostBottom, MostLeft, MostRight, Client, Contents,
5       Center, VertCenter, HorzCenter, Horizontal,
6       Vertical, Scale, Fit, FitLeft, FitRight);
7 ...
```

Listado B.8 Definición del tipo enumerado `TAlignLayout`

De forma parecida, también nosotros podemos definir nuestros propios tipos enumerados, utilizándolos para establecer los valores que se permitirían en variables declaradas con ese nuevo tipo. Los tipos de datos han de definirse en la sección `type`, antes de que puedan ser usados en la sección `var`.

[7]Esta es una técnica útil fundamentalmente cuando se necesita introducir en una cadena caracteres no imprimibles, como pueden ser los tabuladores, saltos de línea y similares.

Suponiendo que estuviésemos trabajando en un proyecto en el que existen varias categorías de producto, podríamos definir un tipo enumerado para identificar cada categoría de forma única. Por ejemplo:

```
...
type
  TProductCategory = (Snacks, Crunchies, Smoothies);
...
var
  aProductCategory: TProductCategory;
...
```

Listado B.9 Definición de un tipo enumerado propio

El mecanismo *Code Insight* del editor de Delphi ofrecerá una lista con los valores permitidos para cualquier variable de un tipo enumerado, como puede apreciarse en la Figura B.3. Internamente, tal y como se observa en la lista (zona inferior derecha de la misma imagen), se asocia un valor numérico a cada uno de los elementos que forman parte de la enumeración. Por defecto el primer elemento tendrá el valor 0, y los valores sucesivos se usarán para las siguientes constantes.

Figura B.3 LOS ELEMENTOS DEL TIPO ENUMERADO TIENEN ASOCIADO UN VALOR

Delphi permite la asignación de valores numéricos concretos a cada uno de los elementos de un tipo enumerado en el momento de la definición, algo que

resultará útil siempre que tenga algún significado para nosotros. La sintaxis a utilizar es la mostrada a continuación:

```
...
type
  TSnacksCategory = (
     Almonds = 10, Crackers = 8, Muffins = 5);
...
```

Listado B.10 Asignación de valores a los elementos de la enumeración

La interpretación que se dé a estos valores será una decisión dependiente de cada aplicación. Podrían utilizarse, por ejemplo, como un indicador de lo saludable que es cada producto en un ranking de 0 a 10.

Enumeraciones con ámbito

Por defecto los identificadores usados como nombres de los elementos de una enumeración se añaden al ámbito global de la aplicación. Esto implica que esos nombres puedan ser usados como tales, sin necesidad de prefijo alguno:

```
aMunchCategory := Snacks;
```

Cuando se hace referencia a un módulo desde otro, los identificadores de las enumeraciones definidas en la sección de interfaz del primero aparecerán como constantes públicas en el segundo. Dado que un programa puede contener referencias a multitud de módulos, la cantidad de identificadores globales podría causar colisiones de nombres[8]. Para evitar este problema existen las enumeraciones con ámbito.

Una enumeración con ámbito no hace públicos sus identificadores, demandando la inclusión del nombre completo del tipo enumerado como prefijo. En

[8]Esta es la razón para el uso de prefijos en casi todas las enumeraciones definidas en las bibliotecas RTL y VCL. Los identificadores pertenecientes a `TAlignLayout`, por ejemplo, inician todos su nombre con el prefijo `al`, como en `alNone`, `alTop`, etc. Esto también se aplicaba a la biblioteca FMX en versiones previas.

consecuencia, tendríamos que usar `TProductCategory.Snacks` en lugar de `Snacks`. Al no ser los nombres de los elementos accesibles desde fuera del tipo enumerado propiamente dicho, no hay necesidad de añadir un prefijo a cada uno de los nombres.

La definición de enumeraciones con ámbito se controla con la directiva de compilador `SCOPEDENUMS`. Esta puede encontrarse en dos posibles estados: `ON` u `OFF`, siendo el último su valor por defecto. Podemos activar temporalmente esta características para nuestros tipos enumerados tal y como se muestra a continuación:

```
1 ...
2 type
3 {$SCOPEDENUMS ON}
4   TSmoothiesCategory = (Kiwi, Pineapple, Chocolate);
5 {$SCOPEDENUMS OFF}
6 ...
```

Listado B.11 Título

De acorde a la anterior definición de tipo enumerado, los identificadores `Kiwi`, `Pineapple` y `Chocolate` no podrán utilizarse sin incluir como prefijo el nombre `TSmoothiesCategory`.

> **NOTA**
>
> *Desde la versión XE6 de Delphi la biblioteca de componentes FMX utiliza por defecto enumeraciones con ámbito, un cambio con respecto a versiones previas. Debes tener este hecho en cuenta, dependiendo de la versión del producto que estés utilizando.*

B.2.6 Subrangos

En ocasiones podemos necesitar un tipo de dato numérico que no sea tan extenso como los tipos de datos básicos mencionados con anterioridad, como `Integer`,

pero que tampoco sea tan limitado como lo es una enumeración[9]. En estos casos un tipo subrango podría ser lo más apropiado.

Imaginemos que queremos asociar un año de fabricación a cada uno de los productos vendidos por una empresa. Un rango razonable para este dato podría ser de 1900 a 2100, suponiendo que no hay objetos previos a 1900 para ser vendidos y que nuestro programa no estará en uso para el año 2100. Podemos definir y utilizar un tipo subrango para este escenario con el código siguiente:

```
1  ...
2  type
3    TManufacturingYear = 1900..2100;
4
5  var
6    manufactured: TManufacturingYear;
7
8  begin
9    manufactured := 1880; // Esto provocará un error
10 ...
```

Listado B.12 Definición de un tipo subrango

El subrango se define estableciendo sus límites, valores mínimo y máximo (ambos incluidos en el subrango), separados por dos puntos. Aunque en este ejemplo los límites son valores numéricos literales, podrían también ser caracteres o incluso identificadores de un tipo enumerado, dependiendo de nuestras necesidades.

Como puede comprobarse en la Figura B.4, la última sentencia provoca un error de compilación porque el valor que estamos intentando asignar está fuera del subrango. En caso de que el valor asignado fuese leído de otra variable, u obtenido como resultado de la evaluación de una expresión aritmética, el compilador no podría detectar el problema, ya que no sabe cuál sería el resultado hasta que se ejecutasen las sentencias previas.

[9]Una enumeración puede alojar un gran número de valores, pero dado que es necesario asignar un nombre distinto a cada uno de ellos, no parece muy razonable tener cientos o miles de ellos.

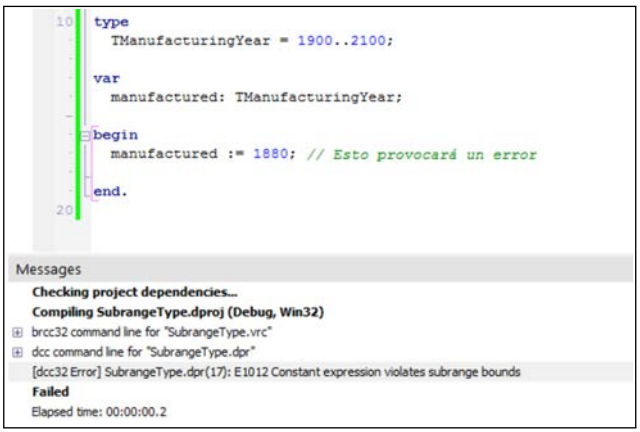


Figura B.4 EL VALOR ESTÁ FUERA DE LOS LÍMITES DEL SUBRANGO

Por defecto el compilador de Delphi no incluye código para comprobar los límites de subrangos en el ejecutable final, pero esta es una característica que puede activarse mediante la directiva `$R+`. Añadirla al código sería equivalente a abrir la ventana **OPTIONS** del proyecto y activar la opción **RANGE CHECKING** de la sección **COMPILER—COMPILING—RUNTIME ERRORS** (véase la Figura B.5).

B.2.7 Vectores y matrices (*Arrays*)

Los descritos en las secciones anteriores son tipos de datos simples, en el sentido de que están diseñados para almacenar un único valor. Cada vez que se asigna un nuevo valor a una variable de un tipo simple, su contenido previo se pierde. Los tipos compuestos (o tipos de datos estructurados), por el contrario, están hechos de tipos de datos simples y tienen la capacidad de contener varios valores de forma simultánea. Dichos valores pueden ser todos del mismo tipo básico o bien una combinación de tipos distintos.

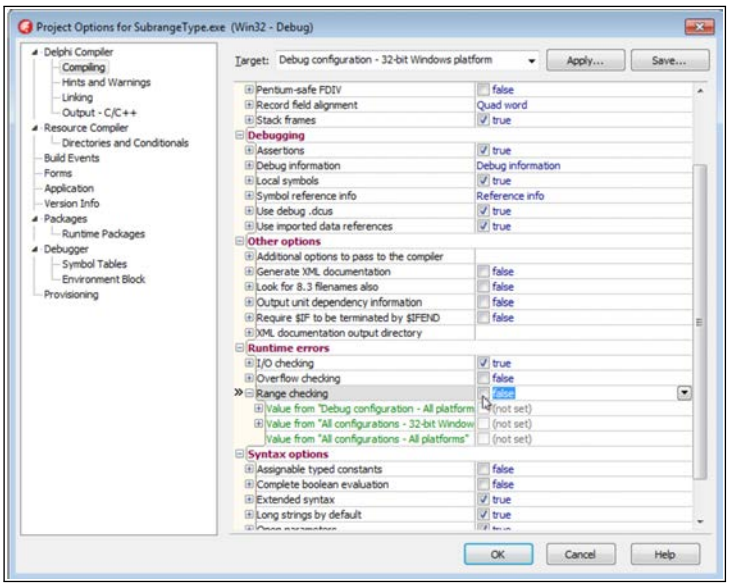


Figura B.5 LA OPCIÓN **RANGE CHECKING** ESTÁ DESHABILITADA POR DEFECTO

El tipo de dato compuesto de uso más habitual es conocido como *array* o matriz[10]. Se trata de una colección de valores del mismo tipo de dato básico. Es posible acceder a cada uno de los elementos en dicha colección mediante uno o más índices, dependiendo del número de dimensiones que tenga el *array*.

Si el número de dimensiones, y el número de elementos existentes en cada dimensión, son establecidos en el momento de la declaración tendremos una matriz estática. También es posible especificarlos a posteriori, en cuyo caso trabajaríamos con una matriz dinámica. Por tanto, si no conocemos de antemano (en el momento en que estamos escribiendo el código) cuántos elementos van a necesitarse, o cuántas dimensiones habrá, siempre elegiremos una matriz dinámica.

[10]Los términos *array* y *matriz* son denominaciones genéricas, existiendo otras más específicas como vector para referirse a las matrices de una sola dimensión (estructura lineal), tabla cuando se hace referencia a matrices bidimensionales, etc.

Declaración de matrices estáticas

Para declarar una matriz estática hay que colocar la palabra reservada `array` delante del tipo de dato básico de cada uno de sus elementos. Tras esa palabra clave, entre corchetes, se especificarán los límites de cada una de las dimensiones siguiendo la sintaxis usada para los subrangos.

Imaginemos que necesitamos almacenar temporalmente en memoria las resoluciones de pantalla contempladas por un grupo de dispositivos y ordenadores. Podríamos utilizar una matriz de cadenas de caracteres, guardando en cada cadena todos los datos de cada resolución específica. También sería posible definir una matriz bidimensional con elementos numéricos, de forma que cada elemento de la primera dimensión contuviese tres elementos: el número de píxeles en horizontal y vertical y el número de colores. El código siguiente muestra cómo declarar y utilizar dos variables según esos dos enfoques:

```
...
var
  res1: array[1..10] of String;
  res2: array[1..10,1..3] of Integer;

begin
  res1[1] := '256x192x16';

  res2[1][1] := 256;
  res2[1,2] := 192;
  res2[1,3] := 16;
...
```

Listado B.13 Declaración de matrices estáticas

Como puede apreciarse en las tres últimas sentencias, a la hora de acceder a un elemento en una matriz multi-dimensional es posible colocar los índices todos juntos, con una sola pareja de corchetes, o bien de manera independiente, cada uno de ellos en su propia pareja de corchetes. Podemos usar la sintaxis que más cómoda nos resulte.

La primera matriz declarada en el código previo, `res1`, tiene diez elementos, con índices del `1` al `10`, y cada uno de ellos contiene una cadena de caracteres.

La segunda variable es una matriz con 30 enteros, estructurados como 10 filas de 3 columnas o viceversa, dependiendo de cómo se interpreten los índices.

Matrices con índices no numéricos

Una particularidad del lenguaje Delphi es que los límites de los subrangos no han de ser necesariamente números enteros. También pueden utilizarse en ese contexto caracteres, así como identificadores pertenecientes a un tipo enumerado. Esos mismos valores podrían ser utilizados para acceder a los elementos de la matriz. Si existe más de una dimensión, el tipo del índice usado en cada una puede ser distinto. Un ejemplo de esta situación sería el mostrado en el código siguiente:

```
1  ...
2  type
3    TMonth = (Jan, Feb, Mar, Apr, May, Jun,
4              Jul, Aug, Sep, Oct, Nov, Dec);
5
6    TContent = (Analysis, Listing, Practical);
7    TPubYear = 1970..1995;
8
9
10 var
11   magazine: array[TPubYear, TMonth, 1..500] of TContent;
12
13 begin
14   magazine[1982, TMonth.Jan, 10] := TContent.Listing;
15 ...
```

Listado B.14 Matrices con índices no numéricos

La variable magazine es una matriz que tiene tres dimensiones. La primera utiliza como índice el subrango TPubYear, y por tanto tendrá como límite inferior el valor 1979 y como límite superior el valor 1995. Esto hace un total de 26 elementos. La segunda dimensión tiene como índice un tipo enumerado, TMonth, que da como resultado un total de 12 elementos para cada año. Finalmente, la tercera dimensión tiene 500 elementos que hacen referencia a cada una de las posibles páginas en la revista.

El tipo de dato `TContent` se ha definido con la intención de clasificar el contenido de revistas de informática atendiendo a tres hipotéticas categorías. La última sentencia del código de ejemplo anterior muestra cómo se indicaría que el número de enero de 1982 de una revista, en la página 10, contiene un listado de código.

Cómo trabajar con matrices dinámicas

Si no conocemos de antemano cuántos elementos va a tener una matriz, porque su estructura solo pueda ser determinada durante la ejecución del programa, tendremos que recurrir a una matriz dinámica. No obstante, este tipo de matrices imponen algunas restricciones, como el hecho de que los índices hayan de ser necesariamente números enteros o que el límite inferior sea siempre cero.

Para declarar y utilizar una matriz dinámica, el primer paso consistirá en declarar adecuadamente la variable que actuará como tal. Debe incluirse la palabra reservada `array`, pero sin especificación de límites. Más adelante, una vez que sepamos cuántos elementos son necesarios, se llamará al método `SetLength` para establecer el tamaño de la matriz. El código siguiente ilustra esta técnica:

```
1  ...
2  var
3    res: array of String;
4  ...
5
6  begin
7    ...
8    SetLength(res, 7);
9    res[0] := '256x192x16';
10 ...
```

Listado B.15 Declaración y uso de matriz dinámica

En este ejemplo la variable `res` tiene 7 elementos, con índices del `0` al `6`. También sería posible trabajar con matrices dinámicas multi-dimensionales, como se muestra en el siguiente ejemplo:

```
var
  numIssues, numPages: Integer;
  collection: array of array of TContent;

begin
  numIssues := 48;
  numPages := 64;

  SetLength(collection, numIssues, numPages);

  collection[0, 0] := TContent.Analysis;
...
```

Listado B.16 Matrices dinámicas multidimensionales

Para declarar la variable `collection` se usa la fórmula `array of array`. Esta puede extenderse, repitiendo la parte of `array` tantas veces como se necesite. Ha de tenerse en cuenta que el procedimiento `SetLength` necesitará tantos parámetros como dimensiones tenga la variable cuyo tamaño va a ajustarse. Cada parámetro establecerá el número de elementos para una de las dimensiones posibles.

Con independencia de que la matriz se haya declarado de manera estática o dinámica, siempre será posible obtener su tamaño actual llamando a la función `Length`. De forma similar, las funciones `Low` y `High` permiten obtener los límites inferior y superior, respectivamente. Al ser utilizadas con matrices multidimensionales, estas funciones devolverán el tamaño o los límites de la primera dimensión. Para obtener datos de la segunda dimensión hay que utilizar la sintaxis `Length(arrayVar[0])`, y lo mismo para dimensiones sucesivas.

B.2.8 Registros

Todos los elementos de una matriz han de ser necesariamente del mismo tipo, por lo que esencialmente son todos idénticos, a pesar de que cada uno de ellos almacene un valor distinto y pueda ser identificado de manera única mediante un índice. En contraposición, un registro es un tipo de dato complejo que se compone de elementos con diferentes nombres, no solo un índice, y distintos

tipos de datos básicos, ya que el hecho de que todos los miembros sean del mismo tipo no es algo obligatorio.

Antes de poder declarar y utilizar una variable de tipo registro es necesario establecer cuál será su estructura y nombre. Es preciso dar a cada uno de los miembros un nombre único, así como indicar cuál será el tipo de la información que almacenará. La sintaxis para hacerlo es relativamente sencilla, como puede verse en el fragmento de código siguiente. Este tiene por objetivo definir un registro para almacenar datos sobre ordenadores antiguos:

```
1  ...
2  type
3    TBitsComputer = (bo8 = 8, bo16 = 16, bo32 = 32);
4
5    TComputer = record
6      name         : String;
7      year         : TPubYear;
8      manufacturer : String;
9      model        : String;
10     bits         : TBitsComputer;
11     RAM          : SmallInt;
12     ROM          : SmallInt;
13     itWorks      : Boolean;
14   end;
15  ...
```

Listado B.17 Definición de la estructura de un registro

Un registro se define de manera similar a la declaración de una lista de variables individuales, pero esto no implica que se asigne memoria alguna para su almacenamiento. Todo lo que hacemos en ese momento es determinar la estructura del registro, no estamos declarando variables. La memoria será asignada una vez que se declare una variable de tipo `TComputer`, usando para ello la sintaxis habitual que ya conocemos.

A diferencia de las matrices, el método de acceso a los miembros de un registro no se lleva a cabo mediante índices, sino con la notación que ya hemos usado muchas veces para hacer referencia a propiedades y métodos de un objeto.

Tenemos que utilizar la sintaxis `variable.miembro`, como se aprecia en la Figura B.6.

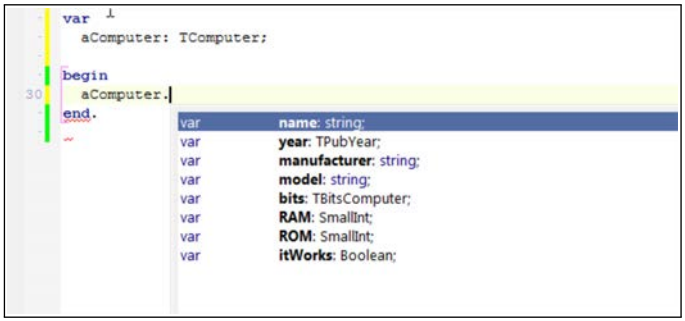


Figura B.6 CODE INSIGHT MUESTRA LOS MIEMBROS DEL REGISTRO

El tipo de la variable `aComputer` es `TComputer` y, por tanto, tiene todos los miembros que se definieron p reviamente en ese tipo de dato. Este hecho puede observarse en la lista de miembros que ofrece *Code Insight*.

Una variable como `aComputer` nos permitiría guardar toda la información relativa a un ordenador. Supongamos, sin embargo, que tenemos una gran colección de ordenadores y que queremos almacenar información sobre todos ellos. Una solución podría ser declarar una matriz de elementos `TComputer`, como se hace a continuación:

```
...
var
  collection: array of TComputer;
  numComputers: Integer;

begin

  // Establecer el número de ordenadores en la colección
  numComputers := 75;
  SetLength(collection, numComputers);

  collection[0].name := 'Atari 400';
```

```
13   collection[0].year := 1979;
14
15 ...
```

Listado B.18 Matriz de registros

El valor de la variable `numComputers` ha de ser establecido antes de que la matriz `collection` pueda ser usada. Su tamaño podría extenderse más adelante si fuese necesario.

Trabajar con variables de tipo registro puede llegar a ser algo tedioso, dado que cualquier referencia a uno de los miembros ha de ir necesariamente precedida por el nombre de la variable. Esa parte ha de ser reiteradamente escrita, tal y como se ve en el código siguiente:

```
1 ...
2   collection[0].name := 'Atari 400';
3   collection[0].year := 1979;
4   collection[0].manufacturer := 'Atari';
5   collection[0].bits := bo8;
6   collection[0].RAM := 48;
7 ...
```

Listado B.19 Acceso a miembros de la matriz de registros

Para evitar tener que repetir el nombre de la variable una y otra vez, podemos recurrir a la sentencia `with`. Su sintaxis queda reflejada en el siguiente fragmento de código:

NOTA

Los miembros de un registro pueden ser de cualquier tipo, incluyendo matrices e incluso otros registros. Cuanto más complejos sean los tipos de los miembros, tanto más difícil será escribir las referencias necesarias para acceder a su contenido. Además, Delphi contempla la definición de propiedades y métodos dentro de los registros, de manera análoga a las clases.

B.2.9 Conjuntos

Las matrices y registros son dos tipos de datos que podemos encontrar en muchos otros lenguajes de programación, aunque en algunos de ellos los registros sean conocidos con otro nombre, como por ejemplo estructuras (`struct` en C/C++). Delphi también cuenta con un tipo de dato no tan común, llamado conjunto.

Una variable de tipo conjunto, como su nombre implica, es capaz de alojar un conjunto de elementos de acuerdo al concepto matemático de conjunto. Todos los elementos de un conjunto son del mismo tipo. El conjunto puede estar vacío. Con un conjunto pueden usarse operadores para comprobar si un cierto elemento está contenido en él, calcular la unión, diferencia e intersección de dos conjuntos, etc.

Habitualmente el tipo de dato base, de cada elemento, de un conjunto es una enumeración o un subrango. En cualquier caso, el conjunto no puede alojar más de 256 elementos. Podríamos usar este tipo de dato para almacenar las conexiones de entrada/salida que ofrece cada dispositivo en una colección, por poner un ejemplo. Esto nos permitiría responder a preguntas del tipo ¿tiene este dispositivo salida para TV?, ¿cuenta con conector para auriculares?

La sintaxis para declarar una variable de tipo conjunto es `set of tipo`, donde tipo puede ser un subrango o una enumeración, así como el nombre de un tipo de dato definido con anterioridad siempre que pertenezca a una de esas dos categorías.

El código siguiente muestra varias de las posibilidades citadas de operaciones sobre conjuntos:

```
1  ...
2  type
3    TConnection = (USB, SD, TV, Headphones, DockStation);
4
5  var
6    connectors: set of TConnection;
7    graphics: set of (sprites, scroll, fonts);
8    TVoutput: Boolean;
9
10 begin
```

```
11
12    // Inicialización del conjunto
13    connectors := [USB, Headphones];
14
15    // Añadir un nuevo elemento
16    connectors := connectors + [TV];
17
18    // Comprobar si un elemento está contenido
19    // en el conjunto
20    TVoutput := TV in connectors;
21  ...
```

Listado B.20 Definición y uso de un conjunto

La variable `connectors` podría estar vacía, podría contener cualquier subconjunto del tipo enumerado `TConnection`, y también podría almacenar todos los valores existentes en el conjunto. Lo mismo sería aplicable a la variable `graphics`, si bien en este caso la enumeración de valores ha sido introducida en la misma declaración del conjunto, en lugar de definirla como un tipo de dato con anterioridad.

Como puede verse en la primera sentencia de asignación del código anterior, podemos establecer el contenido de una variable conjunto a partir de un valor literal[11] de tipo conjunto con los elementos que nos interesen. La variable `connectors` tendrá como valor inicial un conjunto con dos elementos: `USB` y `Headphones`.

A continuación se añade un nuevo elemento al conjunto. En realidad, el operador + lo que hace es crear un nuevo conjunto como resultado de la unión de todos los elementos contenidos en los dos conjuntos que se entregan como parámetros. Finalmente, se usa el operador `in` para comprobar la pertenencia de un elemento al conjunto. Dado que el conjunto contiene el elemento `TV`, la variable `TVOutput` tomaría el valor `True` que sería el resultado generado por la expresión.

[11] Las literales de tipo conjunto se escriben como listas de elementos entre corchetes. Si queremos utilizar el conjunto vacío deberemos asignar el valor `[]`.

B.3 Expresiones

Una vez que conocemos los tipos de datos básicos con que cuenta Delphi, podemos usarlos para crear variables a fin de almacenar temporalmente información en nuestras aplicaciones. Como regla general, estos elementos de datos serán utilizados, en algún momento durante la ejecución del programa, introduciéndolos en expresiones cuyo objetivo será evaluarlos o transformarlos.

En los fragmentos de código usados como ejemplo en las secciones previas, así como en los proyectos desarrollados en los capítulos anteriores, la mayor parte de las expresiones que hemos usado han sido de asignación. La asignación es un tipo de expresión en el que aparece el operador `:=` y en la que se evalúa el operando que está a la derecha para almacenar el resultado obtenido en la variable dispuesta a la izquierda. El operando a la derecha del operador puede ser una constante, otra variable, una expresión aritmética, una expresión relacional, el resultado devuelto por la llamada a un método, etc.

No todos los operadores pueden ser utilizados con todos los tipos de datos existentes en Delphi. En consecuencia, el tipo de una variable actúa en cierta forma como una restricción en cuanto a los operadores que pueden utilizarse con ella y, por tanto, el tipo de expresiones en que podría participar. Además, debe tenerse presente que el mismo operador podría efectuar distintas operaciones dependiendo de los tipos de datos sobre los que se esté trabajando.

Delphi cuenta con los operadores aritméticos, operadores relacionales y operadores lógicos que podríamos encontrar en muchos otros lenguajes de programación. Asimismo también ofrece operadores más específicos, como los que nos permiten operar sobre conjuntos, cadenas de caracteres y punteros.

B.3.1 Expresiones aritméticas, relacionales y lógicas

Aparte de la expresión de asignación, que es posiblemente el tipo de expresión más común y también el más simple en la mayor parte de los lenguajes de programación, muchas de las expresiones restantes pueden ser agrupadas en una de las tres categorías indicadas a continuación:

NOTA

Todos los operadores de Delphi están apropiadamente documentados en la ayuda electrónica del producto. En las secciones siguientes solamente se ofrece una introducción rápida a su finalidad.

- **Aritméticas**: En ellas se usan operadores binarios, lo que significa que se precisan dos operandos. Ambos han de ser de tipo numérico. Además de los cuatro operadores comunes: `+`, `-`, `*` y `/`, también pueden usarse los operadores `div` y `mod`. El primero devuelve el cociente de una división entera, mientras que el segundo calcula el resto de una división entera.

- **Relacionales**: Una expresión relacional se compone utilizando operadores cuya finalidad es comparar los operandos facilitados, determinando si son iguales, distintos, el primero es menor que el segundo, etc. Los operandos pueden ser de tipo numérico, caracteres, booleanos, etc., pero ambos han de ser del mismo tipo. La igualdad se comprueba con el operador = y la desigualdad con el operador <>. Los demás operadores relacionales son los mismos que usaríamos en matemáticas: <, >, <= y >=.

- **Lógicas**: Estas expresiones son también conocidas como expresiones booleanas, dado que los operandos que participan en ellas siempre son de tipo `boolean`. Tenemos los tres operadores lógicos clásicos: `not`, `and` y `or`, así como el operador `xor` (or exclusivo). El primero es unario, solamente necesita un operando dispuesto a la derecha. El resto son binarios.

Aparte de conjuntamente con operadores aritméticos, también es posible usar operandos numéricos con operadores de manipulación de bits. Los nombres de estos operadores son los mismos que los de los operadores lógicos, pero trabajan sobre bits individuales de los operandos en lugar de sobre el valor global contenido en ellos. En este categoría hay dos operadores adicionales, llamados `shl` y `shr`, cuya finalidad es desplazar los bits de un número entero hacia la izquierda o hacia la derecha, respectivamente. El segundo operando usado con estos operadores indica cuántas posiciones serán desplazados los bits del primer operando.

B.3.2 Expresiones con conjuntos

A pesar de que los operadores utilizados en expresiones con conjuntos son los mismos que ya conocemos, como los operadores relaciones y aritméticos, la operación que llevan a cabo es específica para ese tipo de dato. Por ejemplo, ya hemos visto antes que el operador + puede utilizarse para obtener la unión de dos conjuntos. Análogamente, el operador `{` obtiene la diferencia entre dos conjuntos y el operador `*` calcula su intersección.

Gracias a los operadores relacionales es posible comprobar la igualdad y desigualdad entre dos conjuntos, como con cualquier otro tipo de dato. El operador <= nos permite saber si un conjunto es subconjunto de otro, mientras que el operador >= hace la comprobación opuesta, si un conjunto es superconjunto de otro. Todos ellos operan siempre sobre dos conjuntos. La única excepción es el operador `in`. Este necesita un elemento y un conjunto, comprobando si el primero pertenece (está contenido) o no al segundo.

B.3.3 Expresiones con punteros

Al tratar los punteros anteriormente, en una de las secciones precedentes, supimos que los operadores básicos para trabajar con ellos son dos: `@` y `^`. El primero se usa para obtener la dirección donde está almacenado el contenido de una variable, dirección que puede ser guardada en una variable de tipo puntero. El segundo permite acceder al dato al que está apuntando el puntero.

Los punteros pueden compararse, a fin de saber si están apuntando a la misma dirección, usando para ello los operadores de igualdad (=) y desigualdad (<>). Además también es posible usar el operador + para aplicar un desplazamiento sobre la dirección base almacenada en el puntero, por ejemplo para hacer referencia a elementos en una matriz o un registro, asumiendo que el puntero tiene la dirección base de la variable.

B.3.4 Otros tipos de expresiones

Además de las clases de expresiones ya descritas en los puntos previos, hay otras en las que los operandos son cadenas de caracteres. En este contexto el operador + se utiliza para unir dos cadenas, dando una nueva como resultado. También

se pueden emplear con caracteres y cadenas de caracteres los operadores relacionales, comprobando si dos cadenas de caracteres son iguales o no, o si una es menor o mayor que otra desde un punto de vista lexicográfico.

La invocación a un método es también una expresión, en la que una lista de parámetros son entregados como entrada y, opcionalmente, se obtiene un valor de salida como resultado. Los parámetros pueden ser valores simples o bien expresiones, siempre que sus respectivos tipos coincidan con aquellos que espera el método. Estas expresiones serán evaluadas justo antes de llamar al método, enviándose como parámetros los resultados de dicha evaluación. El valor de retorno puede ser asignado a una variable, así como usarse en otra expresión.

Otro tipo de expresión es la indexación de un elemento en una matriz, mediante el operador `[]` descrito con anterioridad. En este caso, el nombre de la matriz y el índice serían los operandos, estableciendo el primero la colección base de elementos y el segundo el desplazamiento a aplicar para acceder a uno de ellos.

Siempre que un cierto valor, ya sea un valor literal, o el obtenido de una variable o como resultado de la evaluación de una expresión, no sea del tipo de dato que se necesita, es posible recurrir a una expresión de moldeado o *casting* para convertirlo al tipo apropiado. Este tipo de expresiones, comúnmente conocidas como moldeado de tipo, sigue la sintaxis mostrada a continuación:

```
newDataType(dataItem)
```

El tipo actual de `dataItem` y el indicado por `newDataType` han de ser tipos de datos compatibles[12]. De no ser así la expresión producirá un fallo, una excepción que podría detener la ejecución del programa. Los punteros son tipos de datos compatibles con otros punteros, aunque sus tipos de dato base (tipo del dato al que apuntan) difieran. Todos los tipos de datos numéricos enteros son también compatibles, como lo son los números enteros y los caracteres.

[12]Si necesitas efectuar una conversión entre tipos de datos que no son compatibles, como podría ser transformar un número en coma flotante a un número entero, deberás usar alguna de las funciones de utilidad que nos ofrece la RTL de Delphi, como podría ser en este caso `Trunc` o `Round`. Estas se encargarán de realizar la adaptación adecuada.

B.4 Sentencias

Tipos de datos y variables, conjuntamente con los operadores que permiten constructor expresiones usando esos datos, forman dos de los pilares del lenguaje Delphi. El tercero lo componen las sentencias ofrecidas por el lenguaje, pensadas para controlar el flujo del programa de distintas maneras, realizar llamadas a métodos, etc.

Las sentencias con que cuenta Delphi son muy parecidas a las que encontramos en la mayoría de lenguajes de programación imperativos, si bien la sintaxis puede diferir ligeramente en algunos casos. Por ejemplo, en Delphi la clásica instrucción condicional `if` debe ir seguida de la palabra reservada `then`. Tras ella se introducirían las sentencias a ejecutar en caso de la expresión evaluada diese como resultado `True`. Esta palabra reservada ya no se usa en muchos de los lenguajes más populares, con la excepción de aquellos que derivan de BASIC.

En Delphi la sentencia más simple que podemos escribir es posiblemente la de asignación, una expresión con dos operandos y un operador. Es una sentencia en la que se lee un valor del operando que está a la derecha para almacenarlo en el operando de la izquierda. La hemos usado repetidamente en capítulos previos para modificar propiedades.

Habitualmente, con la excepción de la propia asignación, una expresión no forma por sí misma una sentencia. El lenguaje necesita saber qué ha de hacer con el resultado producido por la evaluación de la expresión, independientemente de que sea aritmética, relacional o lógica.

B.4.1 Estructuras condicionales

En Delphi tenemos dos estructuras condicionales fundamentales: `if-then-else` y `case`. Con ellas es posible ejecutar o no una sentencia dependiendo de un valor de tipo `boolean`. Este valor puede ser literal, una variable, una constante, una expresión relacional o lógica, el resultado de llamar a un método, etc. Al final, lo que importa es que el valor sea del tipo correcto, `boolean` en este caso. Esto mismo sería aplicable a todas las demás sentencias.

La sintaxis básica de la estructura condicional `if-then-else` es la indicada a continuación:

```
if boolean-value then
  sentencia-a-ejecutar-si-True
else
  sentencia-a-ejecutar-si-False;
...
```

Listado B.21 Estructura de la sentencia condicional `if-then-else`

La parte `else` es opcional. En caso de que necesitemos introducir más de una sentencia en cualquiera de los dos caminos posibles, tendremos que utilizar una sentencia compuesta. Una sentencia compuesta es un bloque de sentencias delimitados por las palabras reservadas `begin` y `end`. Esto también se aplica a otras sentencias Delphi que veremos después, como pueden ser `case`, `while` y `for`.

> **NOTA**
>
> *En Delphi el punto y coma se usa solo para marcar el final de una sentencia completa. En relación a la estructura `if-then-else`, la sentencia completa termina con la sentencia que sigue a la parte `else`, no con la dispuesta tras la palabra reservada `then`. En consecuencia, no se permite el punto y coma justo antes del `else`. Es un error común añadir mecánicamente el punto y coma al final de cada línea cuando uno está acostumbrado a lenguajes como C/C++ o Java.*

La sentencia `case` tiene la sintaxis mostrada a continuación:

```
case referenceValue of
  valor1: sentencia1;
  valor2: sentencia2;
  ...
end
```

Listado B.22 Estructura de la sentencia condicional `case`

Tras la palabra reservada case se ha de entregar un valor de referencia. Este será comparado con los valores[13] enumerados en las líneas siguientes, ejecutándose la sentencia asociada al valor que coincida. Opcionalmente puede añadirse un apartado `else` al final, incluyendo la sentencia a ejecutar si no hay ninguna coincidencia con la lista de valores.

B.4.2 Estructuras iterativas

Las estructuras iterativas, o bucles, se utilizan siempre que es necesario repetir la ejecución de una sentencia o bloque de sentencias acorde al resultado de una expresión. Esta puede ser evaluada al inicio o al final de cada iteración. Dependiendo de lo que se necesite, podemos usar las instrucciones `while`, `repeat` y `for`.

En una sentencia `while` la expresión que se entrega como argumento se evalúa al inicio de cada ciclo. Si el resultado de esta evaluación es `True`, se ejecuta la sentencia que sigue a la palabra reservada `do`. Este proceso se repetirá hasta que la expresión devuelva `False`. A continuación se muestra la sintaxis de esta estructura iterativa:

```
1 while expresión-booleana do
2   sentencia;
3 ...
```

Listado B.23 Estructura de la sentencia iterativa `while`

A diferencia de la anterior, la instrucción `repeat` comienza ejecutando la sentencia asociada, dejando la comprobación de la expresión booleana facilitada tras la palabra reservada `until` para el final. En consecuencia la sentencia que hay dentro de la estructura repetitiva será ejecutada al menos una vez. Las palabras clave `repeat` y `until` forman por sí mismas una sentencia compuesta, por lo que es posible repetir varias sentencias sin necesidad de un bloque `begin-end`. Por ejemplo:

[13] Esos valores no están limitados a tipos numéricos enteros y caracteres, como ocurre en otros lenguajes de programación, permitiéndose el uso de subrangos, así como de listas de valores.

```
repeat
  sentencia;
  ...
until expresión-booleana;
...
```

Listado B.24 Estructura de la sentencia iterativa `repeat`

En lugar de evaluar una expresión de tipo relacional o lógica, como hacen las sentencias anteriores, la instrucción `for` da lugar a bucles en los que el número de iteraciones lo determina la comprobación de si el valor de una variable de tipo ordinal (entero, carácter o enumeración) ha alcanzado un cierto umbral. Este umbral puede establecerse con un número específico, ya sea un valor literal, una variable o el resultado de la evaluación de una expresión aritmética, o bien ser inferido a partir del número de elementos que contiene una colección facilitada como argumento.

El siguiente fragmento de código muestra la sintaxis a utilizar en el primero de los casos:

```
for variable := valorInicial {to|downto} umbral do
  sentencia-a-ejecutar;
...
```

Listado B.25 Sintaxis básica de la sentencia iterativa `for`

Cuando el valor inicial es menor que el umbral se usa la palabra reservada `to` para separarlos, de lo contrario habrá que utilizar la palabra reservada `downto`. En cualquier caso, la sentencia dispuesta tras el do será ejecutada una vez por cada valor en el rango desde `valorInicial` a `umbral`, incluyendo los valores extremos. Como es habitual, la sentencia puede ser un bloque de sentencias delimitado por las palabras clave `begin` y `end`.

Si se utiliza la instrucción `for` para iterar por los elementos contenidos en una colección, como puede ser una matriz, un conjunto, una cadena de caracteres o algún otro tipo especializado de colección, la sintaxis aplicable es la mostrada a continuación:

```
...
for variable in colección do
  sentencia-a-ejecutar;
...
```

Listado B.26 Sintaxis de la sentencia `for` para recorrer colecciones

En este caso la sentencia será ejecutada tantas veces como elementos haya en `colección`, y `variable` iría tomando en cada iteración el contenido del elemento correspondiente. Como es obvio, el tipo de dato de `variable` ha de ser el mismo tipo de los elementos de `colección`.

Todas las construcciones iterativas permiten utilizar la instrucción `break` para interrumpir el proceso repetitivo, con independencia del resultado que devuelva la expresión de control. El flujo de ejecución saltaría a la primera sentencia que haya tras el final del bucle. También puede usarse la instrucción `continue`, dentro del cuerpo del bucle, para evitar la ejecución de las sentencias desde el punto actual hasta el final del bloque, iniciando un nuevo ciclo de repetición.

B.4.3 Procedimientos y funciones

En Delphi las sentencias pueden aparecer en la sección principal de un programa, que es el bloque del módulo `program` comprendido entre las palabras reservadas `begin` y `end`. Sin embargo, la mayoría de las veces las sentencias serán introducidas en el cuerpo de métodos, funciones y procedimientos, definidos en la sección `implementation` de módulos estándar de código.

Procedimientos y funciones solo difieren en un aspecto clave, las últimas devuelven un valor como resultado, mientras que los primeros no lo hacen. Ambos son bloques de código a los que se asocia un nombre[14], conteniendo declaraciones de variables y sentencias ejecutables. Su sintaxis general es la mostrada a continuación:

[14]También es posible crear métodos anónimos, bloques de código a los que no se da un nombre, como tendremos ocasión de comprobar en una sección posterior de este mismo apéndice.

```
1  procedure nombre[(parámetros)];
2  [var declaración-de-variables;]
3
4  begin
5    sentencias;
6    ...
7  end;
8
9  ...
10
11 function nombre[(parámetros)]: tipo-valor-de-retorno;
12 [var declaración de variables;]
13
14 begin
15   sentencias;
16   ...
17   Result := valor; // Debe devolverse un resultado
18 end;
```

Listado B.27 Sintaxis general de definición de procedimientos y funciones

El nombre de un procedimiento (función) ha de seguir las reglas de notación que se describieron en una sección anterior. En cualquier caso, su nombre no debe entrar en conflicto con otros identificadores ya definidos en el ámbito actual, con una excepción: varios métodos pueden utilizar el mismo nombre, viviendo en el mismo ámbito, siempre que reciban listas de parámetros distintas. Esta técnica, conocida como sobrecarga de métodos, es útil para compartir un mismo nombre entre métodos que, realizando la misma tarea, actúan sobre operandos de distintos tipos.

Las variables que se declaran en el interior de un procedimiento son locales a este. Esto significa que se asigna memoria para ellas, y se lleva a cabo su inicialización, cada vez que el flujo de ejecución del programa entra en el cuerpo del procedimiento. De la misma forma, las variables son destruidas una vez que el procedimiento llega a su fin. En consecuencia no es posible acceder a las variables locales desde fuera del propio procedimiento.

Si lo que estamos codificando es una función, tras su nombre y la lista de parámetros hemos de indicar el tipo del valor de retorno. Dicho valor debe ser

asignado a la pseudo-variable `Result` antes de que la función llegue a su fin. El valor puede ser el resultado de la evaluación de cualquier expresión, así como ser obtenido de recursos externos.

Parámetros por valor y por referencia

Tanto los procedimientos como las funciones pueden opcionalmente tomar una lista de parámetros de entrada. El tipo y nombre de esos parámetros han de ser indicados en forma de lista separada por puntos y comas y encerrada entre paréntesis.

Por defecto los parámetros se reciben por valor. En otras palabras, lo que el método recibirá será una copia de los valores originales, no una referencia a las variables en las que están almacenados dichos valores. Esto implica que el método no podrá modificar el contenido original de esas variables externas. Si estamos interesados en poder tener esa opción, debemos indicarlo en la declaración de los argumentos utilizando la palabra reservada `var`, tal y como se muestra en el código siguiente:

```
1 procedure hypotenuse(C1, C2: Real; var H: Real);
2 begin
3   H := H + C1*C1 + C2*C2;
4 end;
5 ...
```

Listado B.28 Parámetros por valor y por referencia

Para llamar a este método el tercer argumento debe ser necesariamente una variable, mientras que los otros dos pueden ser variables, valores literales, expresiones, etc. Lo que hace el método es acumular en la variable una sucesión de hipotenusas, añadiendo al contenido actual el resultado de una expresión aritmética realizada sobre los otros argumentos. El código siguiente es un ejemplo de cómo se utilizaría el anterior procedimiento:

```
1 var
2   H: Real;
3
4 begin
```

```
5   H := 0.0;
6   hypotenuse(3.2, 4.1, H);
7   hypotenuse(1.8, 5.7, H)
8
9   // H contiene la suma de las dos hipotenusas
```

Listado B.29 Invocación de un método con parámetros por valor y referencia

Parámetros con valores por defecto

Los parámetros de entrada de un procedimiento pueden tener asociado un valor por defecto cada uno. El parámetro correspondiente tomaría ese valor en caso de que al invocar al procedimiento no se facilitase explícitamente otro valor. El valor por defecto para un parámetro se establece mediante el símbolo = en la lista de declaración de argumentos, tal y como se ve a continuación:

```
1  ...
2  procedure newComputer(nombre: String; bits: Byte = 8);
3  ...
```

Listado B.30 Especificar valor por defecto para parámetro de entrada

Al llamar a este método, si no se facilita más que un argumento se asignaría el valor `8` al parámetro `bits`, ya que ese es el valor por defecto que se ha especificado en la lista.

Parámetros de tipo matriz

Los parámetros recibidos por un método pueden ser de cualquier tipo, incluyendo entre ellos las matrices. Un argumento de tipo matriz se declara en la cabecera del método usando la sintaxis `array of`, pero sin indicar el número de elementos.

En el cuerpo del método que recibe un argumento de este tipo se recurriría a funciones como `Length`, `High` y `Low` para saber cuántos elementos contiene la matriz, así como cuáles son los límites inferior y superior. Asimismo, sería posible recorrer los elementos de una matriz de la que no se conocen su tamaño

ni límites usando un bucle `for`, tal y como se demuestra en el siguiente fragmento de código:

```
procedure setRes(resolutions: array of String);
var
  aResolution: String;

begin
  for aResolution in resolutions do
    // Hacer algo con la variable aResolution
end;

...
  setRes(['12x15', '15x32']);
...
```

Listado B.31 Parámetro de tipo matriz y su tratamiento

Observa la última sentencia de este ejemplo, en ella se muestra cómo habría que llamar al método `setRes` utilizando un valor literal que es una matriz. También podría haberse declarado la matriz previamente, siendo posible usar tanto matrices estáticas como dinámicas.

> **NOTA**
>
> *Aparte de variables, en el interior de un procedimiento también se permite la declaración de nuevos tipos de datos. Un tipo de dato definido en el cuerpo de un procedimiento es un tipo de dato local, por lo que no será visible fuera del método. Delphi también contempla la definición de procedimientos anidados, un procedimiento creado dentro de otro procedimiento. Los procedimientos anidados únicamente pueden ser usados desde el cuerpo del procedimiento al que pertenecen.*

B.4.4 Control estructurado de excepciones

Ciertas sentencias podrían estar sujetas a condiciones externas en el momento en que son ejecutadas, pudiendo provocar un fallo del programa a pesar de que sean perfectamente correctas desde un punto de vista sintáctico y funcional. Estas sentencias no generan errores de compilación, siendo su comportamiento el que se espera de ellas siempre que se satisfagan las citadas condiciones de funcionamiento. De no ser así, producirán una excepción.

Por ejemplo, una sentencia que evalúe una expresión aritmética puede generar una excepción de división por cero si no se ha comprobado el divisor. El acceso a un elemento de una matriz puede producir una excepción de índice fuera de rango. Los moldeados de tipo inválidos también son susceptibles de provocar excepciones.

Delphi cuenta con instrucciones específicas para gestionar excepciones, capaz de controlarlas evitando así la interrupción del programa. El primer paso para poder utilizarlas es saber qué sentencias pueden producir una excepción, así como el tipo de la excepción. También es vital saber qué queremos hacer, la respuesta que se dará, en caso de que la excepción llegue a darse.

El bloque de sentencias a proteger (con las operaciones que podrían fallar) ha de ser delimitado entre las palabras reservadas `try` y `except`, mientras que la acción a llevar a cabo en caso de fallo seguiría a la palabra clave `except`. El siguiente ejemplo muestra la sintaxis básica. Su finalidad es controlar un hipotético error al acceder a un elemento inexistente de una matriz:

```
1  ...
2  var
3    index: Integer;
4
5  begin
6
7    try
8      for index := Low(myArray) to High(myArray) do
9        // Esta sentencia puede generar una excepción
10       sentencia-a-proteger;
11   except
12     on EAccessViolation do
```

```
13        // Código para gestionar la excepción
14    end;
15  ...
```

Listado B.32 Control de una hipotética excepción

Las palabras `try` y `except` forma un bloque de código, de forma que las sentencias cuyas excepciones van a controlarse pueden ser escritas directamente entre ellas sin necesidad de una pareja `begin-end`. Tras la palabra reservada `except` han de disponerse una o más cláusulas `on`, asociada cada una de ella a una clase de excepción.

`Exception` es el tipo más genérico de excepción. Todas las demás clases de excepciones, hay varias docenas de ellas, derivan de `Exception`. Una de esas clases es `EAccessViolation`, excepción que se genera cuando un programa intenta acceder al contenido de un área de memoria que no ha sido previamente inicializada. Esto es lo que ocurre cuando se intenta leer un elemento que está fuera de los límites de una matriz.

Nuestro código también puede generar sus propias excepciones mediante una sentencia `raise`. Esta es útil para comprobar si una cierta condición es satisfecha, por ejemplo en el interior de un método justo antes de utilizar los parámetros recibidos, pero delegando la gestión de los problemas en el código que ha realizado la llamada. La sintaxis de esta sentencia es la siguiente:

```
1  ...
2  if no-se-satisface-condición then
3     raise exception;
4  ...
```

Listado B.33 Generación de excepciones

La excepción puede ser de cualquiera de las clases de excepción ya predefinidas, así como de un tipo definido por el usuario, específico para las necesidades concretas de la aplicación.

En ocasiones puede interesarnos ejecutar ciertas sentencias siempre, con independencia de que surjan o no excepciones. Las sentencias dispuestas tras la instrucción `try` no llegarán a ejecutarse si se desencadena una excepción, mientras que las que siguen a la palabra reservada `except` solamente se ejecutarán si

la excepción tiene lugar. Aquellas sentencias que deban ejecutarse en cualquier caso han de colocarse en un bloque `finally`, como se muestra en el siguiente ejemplo:

```
var
  pixels: array of array of Byte;

begin

  SetLength(pixels, 1280, 1024);
  try
    // Sentencias que pueden generar una excepción
  finally
    SetLength(pixels, 1, 1);
  end;
...
```

Listado B.34 Código de finalización protegido contra excepciones

B.4.5 Otras sentencias

Además de las ya descritas en las secciones anteriores, Delphi cuenta con algunas sentencias más que apuntamos aquí. Algunas de ellas, como `goto`, son consideradas obsoletas. Heredada de las primeras versiones del lenguaje Pascal, esta instrucción salta a un punto específico del programa, señalado con una etiqueta, alterando por tanto el flujo normal de ejecución. Su sintaxis es la mostrada a continuación:

```
var
  label aExit;

begin
 ...
if condOfExit then goto aExit;
  // Sentencias a ejecutar si condOfExit no es True
aExit:
```

```
9    // Sentencias a ejecutar a la salida
10 ...
```

Listado B.35 Ejemplo de uso de la sentencia `goto`

La etiqueta ha de declararse de manera parecida a como se declararía una variable, pero utilizando como prefijo la palabra reservada `label` y sin especificar un tipo de dato. A continuación, la etiqueta ha de ser colocada en algún punto del código, seguida de dos puntos.

En general, este tipo de construcción es habitualmente sustituido por sentencias condicionales, pero en ocasiones el uso de `goto` puede resultar más fácil y claro.

Otra instrucción útil de Delphi, ya introducida en una sección previa, es `with`. Su finalidad es asumir en un bloque de código la referencia a uno o más objetos, haciendo así más simple el acceso a sus miembros. Por ejemplo:

```
1 with object1, object2, ... do
2    sentencia;
```

Listado B.36 Sintaxis de la sentencia `with`

B.5 Clases y sus miembros

Cuando se trabaja en un proyecto real con Delphi, las definiciones de tipos de datos, declaraciones de variables y la implementación de procedimientos y funciones no se llevan a cabo de manera aislada, sino conjuntamente como parte de la definición de una clase. La introducción de código en el módulo de programa no es algo habitual, ya que dicho módulo lo mantiene automáticamente el Gestor de proyectos y, como ya sabemos, solamente aloja las sentencias necesarias para poner en marcha la aplicación.

Delphi es un lenguaje orientado a objetos, si bien no todos los elementos de un programa pueden ser tratados como objetos. Por dicha razón, en ocasiones se dice que Delphi no es un lenguaje orientado a objetos puro. Hay varios tipos

de datos fundamentales, como son los números, caracteres y booleanos, que son tipos primitivos del lenguaje, en lugar del resultado de *instanciar*[15] una clase.

En las siguientes secciones se describe la sintaxis a utilizar para definir nuevas clases en Delphi. Aprenderemos a usarla a través del desarrollo de un ejemplo práctico, cuya hipotética finalidad sería facilitar la gestión de datos relacionados con micro-ordenadores clásicos.

> **NOTA**
>
> *A continuación nos concentraremos en cómo aplicar técnicas de orientación a objetos con Delphi, pero los conceptos básicos sobre orientación a objetos, tales como encapsulación, herencia, construcción de objetos y poliformismo, se dan por conocidos. Si no estás acostumbrado a esta terminología, tendrás que recurrir a un texto de introducción general sobre este tema.*

B.5.1 Cómo definir una nueva clase

Al igual que los registros, y en general cualquier otro tipo de dato que no sea ofrecido nativamente por el propio lenguaje, las clases han de ser definidas antes de que podamos utilizarlas para crear objetos. Si pretendemos usar la clase desde otros módulos del proyecto, fuera del módulo de código donde está siendo definida, la definición ha de colocarse en la sección `interface`.

Hemos de distinguir claramente entre la definición de una clase y su posterior implementación. La definición solamente es una enumeración de los miembros con que contará la clase, especificando sus nombres y tipos, junto a cualquier atributo aplicable, como puede ser la lista de parámetros de un método. En la implementación, normalmente aportada en la sección `implementation` del módulo, se ha de escribir el código ejecutable que corresponde a cada método de la clase.

[15] El término *instanciar* hace referencia a la operación de creación de una *instancia* (un objeto) a partir de una clase.

Comenzaremos, por tanto, definiendo la nueva clase. En este punto se establece su nombre. Si la clase es derivada de otra ya existente, también es el lugar en el que se indicaría este hecho. Puesto que la definición de una clase da lugar a un nuevo tipo de dato, esta ha de incluirse en el apartado `type` de la sección apropiada. La sintaxis fundamental es la mostrada en el siguiente fragmento de código:

```
type
TMyClass = class[(BaseClass)]
  declaración-de-variables
  ...
  declaración-de-métodos
  ...
end;
```

Listado B.37 Sintaxis básica para la definición de una clase

La clase ascendiente (o base), en caso de que la nueva derive de otra, ha de especificarse tras la palabra reservada `class`, entre paréntesis. El mecanismo de herencia introduce en la nueva clase todas las características y comportamiento de la ascendiente, sirviendo así como punto de partida para extender o especializar su funcionalidad.

Los miembros indicados en la siguiente lista pueden aparecer en el interior de la definición de una clase:

- **Variables**: En el contexto de una clase las variables son conocidas habitualmente como atributos o campos de datos. La declaración de variables en el interior de una clase sigue exactamente las mismas reglas que hemos aprendido en secciones previas. No obstante, hemos de tener en cuenta que estas variables no existen por sí mismas, de manera aislada, sino que se alojarán en el interior de los objetos instanciados a partir de la clase.
- **Procedimientos/Funciones**: Son conocidos de manera genérica como métodos cuando son incluidos en una clase. Lo explicado en secciones previas, en relación a las listas de parámetros, valores de retorno, etc., también es aplicable en este contexto. Normalmente los métodos de una clase son los únicos autorizados a manipular los atributos asociados a los objetos pertenecientes a ella.

- **Constructores**: Estos son métodos pensados específicamente para inicializar nuevos objetos creados a partir de la clase que está definiéndose. Todos los constructores se designan mediante la palabra reservada `constructor` y su nombre siempre es `Create`. Es posible definir varios constructores para una misma clase, siempre que cada uno tenga una lista de parámetros diferente. Profundizaremos sobre este tema en un punto posterior.

- **Destructores**: Su función es complementaria a las de los constructores, estando a cargo de la liberación de los objetos de la clase. Se designan con la palabra reservada `destructor` y su nombre siempre es `Destroy`. No pueden tomar parámetros de entrada.

- **Propiedades y eventos**: Los componentes Delphi, como los que usamos en los proyectos propuestos como ejercicios, son creados definiendo clases que derivan de clases específicas de la FMX. Aparte de atributos y métodos, las clases de componentes siempre ofrecen propiedades. Una propiedad es una forma de controlar el acceso a los atributos de un objeto, en lugar de permitir el acceso directo a los campos de datos. Asimismo, la mayoría de componentes también cuentan con eventos. Estos son un tipo de miembro que no abordaremos aquí.

B.5.2 Visibilidad de los miembros de una clase

Si bien al definir una clase en la sección `interface` de un módulo la estamos haciendo pública, accesible desde otros módulos, esto no implica que pueda accederse libremente a todos sus miembros. La propia clase puede establecer diferentes niveles de visibilidad, agrupando sus miembros en las secciones `private`, `protected`, public y `published`.

Los miembros privados son visibles únicamente para otros miembros de la misma clase, así como para el resto del código contenido en el módulo donde está definiéndose la clase[16]. Este es el nivel de acceso más restrictivo, aplicándose normalmente a atributos a fin de que no puedan manipularse salvo a través de los métodos que ofrezca la propia clase.

[16]Habitualmente cada clase se define en su propio módulo, por lo que los miembros privados no serían accesibles desde fuera del mismo.

Si nos interesa que un miembro sea accesible también desde las clases que pudieran derivarse de la actual, pero solo en ese caso, el nivel de visibilidad a usar es `protected`. Un atributo protegido siempre estará disponible para el código de la misma clase y también para el resto del código del módulo en que se define, así como para las clases derivadas, sin que importe dónde se definan.

Todos aquellos miembros pensados para ser de uso general deberían aparecer en la sección `public`, siendo visibles desde cualquier otro módulo. Los constructores, con pocas excepciones, son siempre públicos, como también suelen serlo los métodos que no están escritos específicamente para uso interno de la clase. También suelen ser públicos los tipos de datos asociados a la clase, como los registros, tipos enumerados, etc. Los atributos raramente se hacen públicos, ya que la clase podría perder el control sobre la validez de su contenido.

Finalmente, en la sección `published` se publican las propiedades pertenecientes a clases de componentes, con el objetivo de que hacerlas accesibles para los diseñadores integrados en el IDE de Delphi. Los miembros publicados incorporan información adicional que el IDE de Delphi sabe cómo recuperar, permitiendo la manipulación de los componentes durante la fase de diseño, sin que el programa esté en ejecución.

Aunque no es un requisito imprescindible, es posible tener propiedades no publicadas en una clase. Estas no aparecerían en el Inspector de objetos, durante la fase de diseño, pero sí serían accesibles desde el código del programa, por lo que el programador podría usarlas mientras ejecuta la aplicación.

> **NOTA**
>
> *Los identificadores usados en la sección `published` deben ser ASCII, no se permite el uso de UNICODE en ese contexto.*

B.5.3 Construcción de objetos

Cuando se declara una variable de un tipo básico, es el compilador de Delphi el que se encarga de asignar la memoria necesaria para almacenar el valor que puede contener, así como de llevar a cabo la inicialización de su contenido. El compilador conoce la estructura de los tipos de datos ofrecidos por el lenguaje

Delphi, por lo que es capaz de realizar esas tareas. No obstante, esto no se aplica a las clases definidas por el usuario.

La declaración de una variable cuyo tipo es una clase no provoca la instanciación de esta, por lo que no se crea automáticamente ningún objeto, sólo se reserva un pequeño bloque de memoria a fin de almacenar la referencia a cualquier objeto que pertenezca a dicha clase. Por esta razón, el primer paso que se da usualmente es crear un objeto y almacenar la referencia al mismo en una variable, como se muestra a continuación:

```
type
  class TMyClass = class
    ...
  end;

var
  myObject: TMyClass;

begin
  myObject := TMyClass.Create;
...
```

Listado B.38 Creación de un objeto de una clase

El constructor es el encargado de asignar la memoria que se necesite para crear un objeto, almacenando todos los datos asociados al mismo, así como el responsable de llevar a cabo cualquier tarea de inicialización, como la asignación de valores por defecto a los atributos, apertura de archivos, preparación de conexiones, etc. En caso de que la clase no aporte un constructor, el compilador de Delphi añadirá uno por defecto capaz de ocuparse del trabajo esencial.

Una misma clase puede tener más de un constructor, siempre que cada uno acepte una lista de parámetros distinta. De esta forma es posible facilitar más de una vía de inicialización de los objetos, a partir de distinta información. En cuanto se defina un constructor en el interior de una clase Delphi ya no incluirá el constructor por defecto. En consecuencia, si queremos permitir la creación de objetos a partir de la clase sin que se precisen parámetros, deberemos declarar e implementar explícitamente un constructor por defecto.

Si una clase es derivada de otra, la primera debe encargarse de la inicialización de los miembros nuevos que haya añadido, mientras que la segunda continuará como responsable de todo lo que se haya heredado. Este mecanismo necesita nuestra intervención, utilizando el constructor de la nueva clase la palabra reservada `inherited` seguida del nombre del constructor base y los parámetros que precise para poder realizar adecuadamente su trabajo.

B.5.4 La clase TCollectible

Aprovechando lo que hemos aprendido hasta este punto, vamos a definir una nueva clase cuya finalidad sería almacenar información básica relativa a cada uno de los objetos coleccionables existentes en una colección. Iniciamos un nuevo proyecto vacío, al que llamaremos `MicroClassicProject`, y añadimos un nuevo módulo de código a fin de definir en él un a clase a la que llamaremos `TCollectible`.

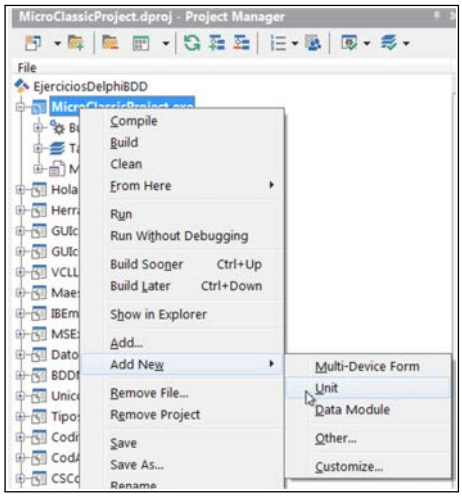


Figura B.7 AGREGAMOS UN MÓDULO DE CÓDIGO AL PROYECTO

Una vez que hayamos creado el proyecto, el primer paso será añadir un nuevo módulo tipo *unit*, un módulo de código en el que escribiremos la definición de la clase. Abrimos el menú contextual del proyecto, en el Gestor de proyectos, y

usamos la opción **ADD NEW—UNIT**, tal y como se muestra en la Figura B.7. Guardamos el módulo con el nombre `Collectible`. Inicialmente su contenido será el que se muestra a continuación.

```
1 unit Collectible;
2
3 interface
4
5 implementation
6
7 end.
```

Listado B.39 Contenido inicial del nuevo módulo de código

El nombre que sigue a la palabra reservada `unit` es el mismo que hemos dado al archivo en que se ha guardado el módulo. Por lo demás solo tenemos una sección `interface` y otra `implementation`, ambas vacías.

Declaración de atributos

Vamos a considerar información básica de cualquier objeto coleccionable los siguientes datos:

- **Nombre**: Una cadena de caracteres conteniendo el nombre completo del objeto, a fin de poder identificar de manera única cualquier objeto de la colección.
- **Año de lanzamiento**: Un número entero para almacenar el año en que se lanzó el objeto, ya sea un ordenador, un software, el año de publicación de una revista, etc.
- **Estado de conservación**: Un subrango definido por el usuario indicando el estado físico en que se encuentra el objeto.
- **Descripción**: Una cadena de caracteres con una caracterización más extensa que la que ofrece el nombre del objeto.
- **Imágenes**: Una matriz conteniendo imágenes relacionadas con el objeto, por ejemplo fotografías de un ordenador, digitalizaciones de la portada de

libros y revistas, capturas de pantalla de software, etc. El tipo base de esta matriz será `TBitmap`, que está definido en el módulo `FMX.Graphics`.

Cada uno de estos elementos de datos aparecerá dentro de la definición de la clase en forma de atributo (variable). Además añadiremos los miembros necesarios para poder acceder a dichos atributos, junto con cualquier otro método que consideremos necesario. Por ejemplo, podría introducirse un método para insertar nuevas imágenes asociadas a un objeto, o un atributo entero para saber cuántas imágenes hay.

Comenzaremos añadiendo a la sección `interface` una cláusula `uses` haciendo referencia al módulo `FMX.Graphics`, ya que vamos a utilizar tipos de datos definidos en el mismo:

```
1 interface
2 ...
3 uses
4   FMX.Graphics; { Módulo con la definición de TBitmap }
```

Listado B.40 Referencia al módulo `FMX.Graphics`

A continuación insertaremos una sección `type` conteniendo la definición de los tipos de dato específicos para nuestra clase. Habrá dos tipos propios: un subrango para el año de lanzamiento y otro subrango para el estado de conservación:

```
1 type
2   TLaunchYear    = 1965 .. 1995;
3   TConsState     =    0 .. 100;
```

Listado B.41 Definición de tipos de datos propios

Esto es todo lo que necesitamos para poder comenzar a definir nuestra clase `TCollectible`. Esta definición se añadirá al apartado `type` de la sección `interface`. Por tanto será accesible desde fuera del módulo actual. De hecho, el objetivo de esta clase será actuar como base de otras, ya que no representa un tipo específico de objeto como podría ser un ordenador, revista o periférico. Más adelante se definiría una clase especializada para cada uno de estos tipos de ob-

jeto, heredando un núcleo común de la clase `TCollectible`. En consecuencia `TCollectible` será una clase abstracta[17].

La definición de la clase comenzará con la declaración de los atributos privados, las variables que servirán para almacenar la información asociada a cada uno de los objetos de la colección:

```
TCollectible = class abstract
  private
    Fname        : String;
    Fyear        : TLaunchYear;
    Fstate       : TConsState;
    Fdescription : String;
    Fpictures    : array of TBitmap;
    nPictures    : Integer;
```

Listado B.42 Atributos de la clase

Declaración de métodos privados

También habrá algunos métodos de uso interno en la clase, por lo que habrá que declararlos como miembros privados. En el interior de la definición de una clase sólo hay que facilitar la declaración de los métodos, mientras que la implementación (el cuerpo de los métodos) aparecerá posteriormente en la sección `implementation`.

La declaración de un método se compone de su nombre, la lista de parámetros que necesita y el tipo de su valor de retorno en caso de que fuese una función. Habitualmente los miembros privados están enfocados a hacer más fácil el acceso a algunas propiedades, así como a llevar a cabo tareas que no tienen porqué ser conocidas por el usuario de la clase.

```
{ Métodos de uso interno }
```

[17]Las clases abstractas se denotan con la palabra reservada `abstract`, que deberemos colocar justo antes del nombre de la clase. El objetivo de una clase abstracta es exclusivamente servir como base de otras, por ello Delphi no nos permitirá crear nuevos objetos a partir de ella.

```
2 procedure setName(name: String);
3 function  getPicture(Index: Integer): TBitmap;
4 procedure extendPictures;
```

Listado B.43 Declaración de métodos privados

Declaración del constructor

La lista de miembros públicos comenzará con la declaración del constructor. Solamente habrá uno, y necesitará como parámetro el nombre del objeto a crear. Su declaración será la siguiente:

```
1 public
2   { Sólo hay un constructor.
3     Necesita el nombre del objeto a registrar }
4   constructor Create(name: String);
```

Listado B.44 Declaración del constructor

Dado que el objeto se construye estableciendo únicamente su nombre, la clase tendrá que ofrecer un mecanismo que permita fijar el resto de atributos, así como obtener su contenido siempre que sea necesario. Hay básicamente dos enfoques posibles para hacerlo, escribiendo métodos con ese objetivo o bien ofreciendo propiedades que actúen como pasarelas de acceso a los atributos. Vamos a usar una combinación de estas dos vías.

Declaración de propiedades públicas

Las propiedades representan un camino natural para acceder a los datos almacenados en un objeto, gracias a la sintaxis `objeto.propiedad`. El usuario de la clase puede tener la sensación de que tiene acceso libre a las variables, dado que puede leer y escribir su contenido sin necesidad de llamar a ningún método. No obstante, es posible definir propiedades solamente de lectura, así como propiedades solamente de escritura si bien este caso es mucho menos común. A pesar de su apariencia, internamente una clase puede controlar el acceso a las propiedades durante el proceso de lectura y de escritura.

Empecemos por el nombre del objeto coleccionable, estableciendo durante su creación. La siguiente propiedad pública permitirá tanto su lectura como la modificación, cambiando el nombre por otro:

```
...
 property Name: String read Fname write setName;
...
```

Listado B.45 Declaración de la propiedad `Name`

El tipo de la propiedad se especifica tras el nombre de la propiedad. Dicho nombre será el que se utiliza para acceder a ella, con independencia de cuál sea el nombre real del atributo en que se almacena la información. La cláusula `read` indica qué debe hacerse cuando se quiera leer el valor de la propiedad. En este caso el valor se obtiene directamente desde el atributo asociado: `Fname`. De manera análoga, la cláusula `write` establece la variable o método a usar cuando se quiera asignar un nuevo valor a la propiedad. El uso de un método permite efectuar cualquier comprobación sobre el valor recibido, así como realizar trabajos adicionales vinculados al cambio del contenido de la propiedad.

El método `setName`, definido previamente en la sección privada, recibirá como parámetro de entrada un `String`, dado que ese es el tipo correspondiente a la propiedad `Name`. Si la cláusula `read` hiciese referencia a un método `getName`, en lugar del atributo, ese método habría que definirlo como una función sin parámetros de entrada y devolviendo un valor de tipo `String`.

Siguiendo el mismo enfoque anterior, se añadirán a la definición de la clase propiedades para acceder a los datos restantes del objeto, haciendo posible la lectura de cada atributo y controlando los cambios sobre ellos. Por simplicidad, en el código de ejemplo siguiente solamente se introducirán las propiedades leyendo y escribiendo directamente de los correspondientes atributos, sin añadir métodos de control:

```
{ Acceso a los demás datos contenidos en el objeto }
property Year : TLaunchYear read Fyear write Fyear;
property State : TConsState read Fstate write Fstate;
property Description : String read Fdescription
                               write Fdescription;
```

```
6 property NumPictures : Integer read nPictures;
7 ...
```

Listado B.46 Definición del resto de propiedades

La propiedad `NumPictures` es solamente de lectura, dado que el número de imágenes cambia indirectamente, cuando se añade una nueva imagen a través del método encargado de dicha tarea. Por tanto, no tiene sentido que pueda cambiarse el contenido de dicho atributo de forma manual. El mencionado método público se introduce en la clase tal y como se muestra a continuación:

```
1 ...
2 procedure addPicture(picture: TBitmap);
3 function hasPictures: Boolean;
4 ...
```

Listado B.47 Métodos públicos asociados a la propiedad `NumPictures`

Llamando al método `hasPictures` el usuario de la clase podrá saber si un cierto objeto tiene asociada alguna imagen. En caso afirmativo, sería interesante contar con algún medio para poder acceder a las mismas. Con este objetivo se añadirá a la clase una nueva propiedad. Esta será algo especial, ya que será una propiedad *indexada*, lo cual significa que se comportará como lo haría una matriz. Lo mostrado a continuación es la declaración de la propiedad:

```
1 ...
2 property Picture[I: Integer] : TBitmap read getPicture;
3 ...
```

Listado B.48 Definición de la propiedad indexada `Picture`

Tras el nombre de la propiedad, encerrado entre corchetes, se establece el nombre y tipo del índice. El método de lectura recibirá ese índice como parámetro de entrada, teniendo que devolver como resultado un objeto de la clase `TBitmap`.

Implementación de métodos

Llegados a este punto la definición de la clase ya está acabada. El paso siguiente será proceder a la implementación de los métodos pertenecientes a la misma, introduciendo su código, conjuntamente con cualquier declaración que pudieran necesitar, en la sección `implementation` de nuestro módulo de código.

El primer método a implementar será el constructor, tal y como se muestra a continuación:

```
...
implementation

const INI_NUM_PICTURES = 1;

{ El constructor precisa el nombre del objeto, e
  inicializa los miembros de datos esenciales }
constructor TCollectible.Create(name: String);
begin
  setName(name);
  SetLength(Fpictures, INI_NUM_PICTURES);
  nPictures := 0;
end;
...
```

Listado B.49 Implementación del constructor

La constante `INI_NUM_PICTURES` establece la capacidad inicial que se asignará a la matriz dinámica `Fpictures`, definida anteriormente como atributo. Esta capacidad, como comprobaremos después, irá duplicándose cuando sea necesario.

Dado que `Create` es un miembro de la clase `TCollectible`, a la hora de escribir la implementación el nombre de la clase ha de aparecer como prefijo. Esta regla se aplica a todos los miembros de una clase, no exclusivamente al constructor. En caso contrario serían tratados como procedimientos y funciones independientes, en lugar de como métodos de la clase.

Aparte de almacenar el nombre del objeto coleccionable que se recibe como parámetro de entrada, el constructor también inicializa el tamaño de la matriz `Fpictures` y el número de imágenes contenido actualmente en dicha matriz.

El siguiente paso será implementar los métodos encargados de leer y escribir los valores de las propiedades. La propiedad `Name` se lee directamente desde el atributo asociado, para modificarla es necesario implementar el método siguiente:

```
{ Método para modificar la propiedad Name }
procedure TCollectible.setName(name: String);
begin
  Fname := name;
end;
```

Listado B.50 Método de escritura de la propiedad `Name`

Como puede apreciarse, no hacemos nada especial en este método. Sencillamente se almacena el nuevo valor en el atributo correspondiente, reemplazando el valor anterior. En la práctica, sin embargo, un método que controla la escritura en una propiedad debería comprobar la validez del nuevo valor, así como notificar el cambio a otros elementos a fin de desencadenar su actualización.

El segundo método vinculado a una propiedad es el método de lectura de la propiedad `Picture`. Esta es una propiedad indexada que devuelve un objeto de tipo `TBitmap` como resultado. Su implementación es la mostrada a continuación:

```
{ Facilitar el acceso a las imágenes del objeto }
function TCollectible.getPicture(Index: Integer): TBitmap;
begin
  if Index < nPictures then
    Result := Fpictures[Index]
  else
    Result := nil;
end;
```

Listado B.51 Método de lectura de la propiedad indexada

En este caso la lectura de la propiedad implica una comprobación previa, verificando que el índice entregado no está fuera de rango. Si el índice es menor que el número de imágenes en la matriz (es una matriz dinámica, por lo que el índice inferior es el 0), se obtiene la imagen correspondiente del atributo `Fpictures` y se devuelve como resultado. En caso contrario se devuelve el valor `nil`. En Pascal `nil` es el equivalente al `NULL` de otros lenguajes, denotando la ausencia de valor alguno. Una alternativa sería que el método desencadenase una excepción, en lugar de devolver `nil`. Para ello usaríamos la instrucción `raise` que ya conocemos.

Finalmente, completaremos la definición de nuestra clase implementando el método público encargado de añadir nuevas imágenes asociadas a los objetos, junto con el método privado que se ocupará de extender el tamaño de la matriz en que se almacenan. El código de esos métodos es el siguiente:

```
1  { Añadir una nueva imagen a la lista de imágenes }
2  procedure TCollectible.addPicture(picture: TBitmap);
3  begin
4    if nPictures = Length(Fpictures) then
5      extendPictures;
6
7    Fpictures[nPictures] := picture;
8    inc(nPictures);
9  end;
10
11 { Extender la matriz duplicando su tamaño }
12 procedure TCollectible.extendPictures;
13 begin
14   SetLength(Fpictures, Length(Fpictures) * 2);
15 end;
```

Listado B.52 Métodos adicionales para la propiedad `Picture`

El método `addPicture` comienza comprobando si el número actual de elementos en la matriz es igual a su tamaño. En dicho caso será necesario ampliar el tamaño de la matriz, una tarea que se delega en el método `extendPictures`. Como puede verse en el código de dicho método, el procedimiento es muy

simple: se invoca al método `SetLength` ajustando el tamaño de la matriz al doble, multiplicando por 2 el valor que devuelve la función `Length`.

> **NOTA**
>
> *El procedimiento `SetLength` preserva el contenido actual de la matriz cuyo tamaño está alterándose siempre que su nuevo tamaño sea superior al actual. En otras palabras, si reducimos el tamaño de la matriz lógicamente parte de su contenido se perderá.*

Una vez que se hemos confirmado que hay espacio suficiente en la matriz, se procede a almacenar en ella la nueva imagen. Dado que el número de imágenes en la matriz ha crecido, tenemos que incrementar el valor actual del atributo `nPictures`. Lo hacemos llamando al procedimiento `inc`, que es útil para incrementar cualquier valor ordinal (entero, enumeración o subrango) en una unidad. Existe un procedimiento complementario, llamado `dec`, que reduce en una unidad el valor de la variable entregada como parámetro.

Nuestra clase no cuenta con un destructor, dado que no necesita liberar explícitamente recurso alguno como podría ser devolver memoria asignada, cerrar archivos o cerrar conexiones de comunicación. Solamente hay un elemento dinámico, la matriz `Fpictures`, pero será liberado automáticamente una vez que no exista referencia alguna a ella. Si hubiésemos usado un puntero `TBitmap`, en lugar de una matriz `TBitmap`, la memoria la habríamos asignado mediante los procedimientos `New` o `GetMem`, teniendo posteriormente que ser librada con una llamada a `Dispose` o `FreeMem`. En esta situación el destructor resultaría imprescindible a fin de evitar pérdidas de memoria.

B.5.5 La clase TComputer

La clase `TCollectible`, definida e implementada en la sección precedente, es una clase abstracta. Esto significa que no podemos crear objetos a partir de esa clase. En consecuencia, a primera vista podría parecer que es una clase sin aplicación práctica alguna. No obstante, el objetivo de esa clase es actuar como base común para otras clases del proyecto. Ella aporta los atributos y los métodos que son comunes a todas las clases descendientes. Las nuevas clases pueden

introducir sus propias propiedades y métodos, convirtiéndose en derivadas especializadas de su clase base. Además, el hecho de que objetos de diferentes clases sean todos descendientes de una clase común permite implementar procedimientos de manera genérica gracias al polimorfismo.

Tomando como base la clase `TCollectible`, la primera clase que vamos a derivar de ella se llamará `TComputer`. Estará especializada en un tipo específico de objeto coleccionable: ordenadores antiguos. Tendremos que añadir un nuevo módulo de código al proyecto, como hicimos anteriormente para la clase `TCollectible`. Este nuevo módulo se llamará `Computer`.

Comenzaremos añadiendo a la sección `interface` una referencia al módulo `Collectible`, y definiendo los tipos de datos que necesitaremos para la clase `TComputer`. En este caso solo se precisará un nuevo tipo enumerado:

```
1 interface
2
3 uses
4   Collectible;
5
6 type
7   TBitsComputer = (bits8, bits16, bits32);
```

Listado B.53 El nuevo módulo hace referencia al anterior

La nueva clase declarará al inicio los atributos privados que necesitará para almacenar información relacionada con ordenadores clásicos. Dado que la clase `TComputer` va a ser derivada de `TCollectible`, todos los atributos básicos de cualquier objeto coleccionable ya estarán ahí. Por tanto solo hay que añadir los atributos específicos:

```
1 type
2 ...
3   TComputer = class(TCollectible)
4    private
5     Fmanufacturer: String;
6     Fmodel       : String;
7     Fbits        : TBitsComputer;
```

```
    FRAM          : SmallInt;
    FROM          : SmallInt;
    Fworks        : Boolean;
    ...
```

Listado B.54 Atributos de la clase TComputer

A continuación se introducirán los miembros públicos de la clase. Habrá un constructor, así como un conjunto de propiedades para acceder a la información del ordenador. En este caso todas ellas van a ser directamente leídas y escritas de sus respectivos atributos, a pesar de que en un situación real normalmente la operación de escritura vendría asociada a métodos que se encargarían de verificar la validez de la información.

El código siguiente completaría la definición de la clase:

```
  ...
  public
    constructor Create(name: String);

    property Manufacturer: String
             read Fmanufacturer write Fmanufacturer;
    property Model: String
             read Fmodel write Fmodel;
    property Bits: TBitsComputer
             read Fbits write Fbits;
    property RAM: SmallInt
             read FRAM write FRAM;
    property ROM: SmallInt
             read FROM write FROM;
    property ItWorks: Boolean
             read Fworks write Fworks;
  end;
```

Listado B.55 Completamos la definición de la clase

Al no existir más elementos adicionales, el único método a implementar es el propio constructor. Al igual que el constructor de `TCollectible`, este recibe

como entrada el nombre del nuevo objeto, una información que ha de almacenarse en el atributo apropiado. Ese atributo fue declarado como miembro privado en la clase `TCollectible`, por lo que no es accesible desde fuera, ni siquiera desde una clase derivada como es `TComputer`. De hecho la clase derivada no tiene por qué saber dónde se aloja esa información, no es su responsabilidad. Todo lo que tiene que hacer es delegar el proceso de inicialización de los miembros en el constructor heredado, acometiendo después la inicialización de los miembros específicos que pertenecen a la nueva clase.

En resumen, el constructor de `TComputer` será implementado como se muestra a continuación:

```
1  ...
2  implementation
3
4  constructor TComputer.Create(name: String);
5  begin
6    inherited Create(name);
7
8    Fbits  := bits8;
9    FRAM   := 64;
10   FROM   := 32;
11   Fworks := True;
12 end;
```

Listado B.56 Implementación del constructor

El constructor base se invoca mediante la palabra reservada `inherited`, entregando los datos que precisa. Una vez que el constructor base ha terminado su trabajo, podemos proceder con la asignación de los valores por defecto a los miembros específicos de nuestra clase. En este ejemplo hemos utilizado valores teóricamente comunes en nuestra colección de ordenadores, asumiendo que la mayoría de ellos son ordenadores de 8 bits con 64 KB de memoria y que están en perfecto orden de funcionamiento.

De manera similar podríamos definir el resto de clases necesarias para el proyecto, por ejemplo `TSoftware`, `TPeripheral`, `TMagazine`, etc. Todas ellas serían derivadas de la clase `TCollectible`. Cada clase tendría algunos

miembros específicos, así como posiblemente métodos diseñados para su comportamiento particular.

B.6 Miembros de clase y el objeto self

Los atributos definidos en las clases anteriores son todos *miembros de instancia*. Esto significa que cada objeto de la clase tendrá su propio conjunto independiente de valores para esos atributos, sin ninguna vinculación con los valores de los demás objetos. Esto es esencial, ya que queremos mantener datos diferenciados para cada objeto, a pesar de la estructura sea la misma.

Como es lógico, no se lleva a cabo una copia del código de los métodos de la clase para cada objeto en particular, sino que este es compartido por todos los objetos de dicha clase. Por esa razón surge la cuestión siguiente: cómo sabe el método invocado sobre qué objeto está trabajando en cada momento. La respuesta la encontramos en la variable `self`, que almacena una referencia al objeto actual (el objeto desde el que se ha efectuado la llamada al método).

Siempre que una sentencia de un método hace referencia a miembros de instancia de la clase, sin usar como prefijo el nombre de un objeto específico, se asume que ese objeto es `self`. La referencia también puede ser explícita, a fin de evitar cualquier tipo de ambigüedad que pudiera haber en caso de que tener dos identificadores idénticos en el mismo contexto. Por ejemplo:

```
1 procedure TComputer.setRAM(FRAM: SmallInt);
2 begin
3   self.FRAM := FRAM;
4 end;
```

Listado B.57 Referencia mediante el objeto `self`

En este caso el parámetro `FRAM` tiene el mismo nombre que el atributo privado `FRAM`, por ello se ha añadido la referencia explícita `self` para distinguir entre uno y otro.

En una clase también pueden existir *miembros de clase*, conocidos asimismo como *miembros estáticos*. A diferencia de los miembros de instancia, los miembros de clase están asociados a la clase propiamente dicha, no a los objetos que se crean a partir de ella. En consecuencia, no es necesario tener una copia de ellos para cada objeto.

Los atributos de clase resultan útiles para compartir información entre los objetos existentes de una clase. Este tipo de atributos se definen en la sección `class var` de la clase. También hemos de tener en cuenta que no podemos utilizar la sintaxis `self.atributo` u `objeto.atributo` para acceder a los atributos de clase, ya que el campo en cuestión no pertenece a un objeto concreto. Por esta razón, la sintaxis que debemos utilizar será `clase.atributo`.

Asimismo, los *métodos de clase* pueden ser ejecutados sin necesitar una referencia a un objeto, mediante la sintaxis `clase.método`. Estos métodos no incorporan la referencia `self`, por lo que no podrán acceder a atributos de instancia a menos que se les facilite explícitamente una referencia a un objeto. Un método de clase se define usando la palabra reservada `class` delante de `procedure` o `function`.

Además de atributos de clase y métodos de clase, en una clase también pueden definirse propiedades de clase, constructores de clase y destructores de clase. Las propiedades son similares a los métodos de clase, pudiendo accederse a ella mediante la sintaxis `clase.propiedad` y no teniendo integrado el objeto `self`. Sencillamente son un atajo para acceder a los atributos de clase. En cuanto a los constructores y destructores de clase, su objetivo es hacer más fácil la inicialización de los atributos de clase, cuando se hace referencia a la clase por primera vez, y el proceso de liberación de recursos, respectivamente.

B.6.1 La clase TCollection

Supongamos que queremos tener en nuestro proyecto una clase que facilite la gestión de todos los objetos existentes en la colección. A pesar de la presencia de muchos objetos diferentes, todos ellos pertenecen a una misma colección.

Puesto que no existe más que una colección (una por cada usuario), no tiene sentido poder tener más de un objeto de esta clase. Esta es una situación bastante habitual, normalmente resuelta mediante el uso de un patrón de diseño llamado *singleton*. Un atributo de clase contendrá la referencia al único objeto existente

de la clase, un método de clase con el nombre `getInstance` devolverá dicha referencia, y un constructor de clase se ocupará del proceso de inicialización.

La nueva clase se llamará `TCollection` y su definición será introducida en la sección `interface` de un nuevo módulo de código que hemos de agregar al proyecto, y al que llamaremos `Collection`. El código siguiente corresponde a esa definición:

```
...
type

TCollection = class
private
  class var refInstance: TCollection;

  { TODO: Añadir los miembros de clase para contener
          los objetos de la colección }

  class constructor Create;

  class destructor  Destroy;

  constructor Create;

public
  class function getInstance: TCollection;
  class property Instance: TCollection read refInstance;
end;
```

Listado B.58 Definición de la clase TCollection

El atributo de clase `refInstance` servirá para almacenar la referencia al único objeto que se creará con esta clase. Es posible agregar miembros adicionales según se precisen, tanto miembros de instancia como miembros de clase, con el objetivo de dar a la clase la funcionalidad que se espera de la misma.

Como puede comprobarse, además del constructor de clase, también se ha definido un constructor estándar. Ambos son privados, como lo es el destructor, y ninguno de ellos toma parámetros de entrada. Los únicos miembros públicos

son la función `getInstance` y la propiedad `Instance`. Ambos tienen el mismo propósito: devolver la referencia que se tiene almacenada en el atributo `refInstance`.

El paso siguiente será implementar el constructor de clase y el destructor de clase. El primero será llamado automáticamente por Delphi, no necesariamente en cuanto se inicie la aplicación, pero siempre antes de que se haya accedido por primera vez a cualquiera de los miembros de la clase. Recuerda que este constructor es un miembro privado, por lo que resulta inaccesible para el usuario de la clase.

El código mostrado a continuación corresponde a la implementación de estos dos miembros:

```
1  ...
2
3  implementation
4
5  class constructor TCollection.Create;
6  begin
7    if refInstance = nil then
8      refInstance := TCollection.Create;
9  end;
10
11 class destructor TCollection.Destroy;
12 begin
13   refInstance.Free;
14   refInstance := nil;
15 end;
```

Listado B.59 Implementación de constructor y destructor

El constructor verifica que la variable `refInstance` está vacía, procediendo en ese caso a crear un objeto `TCollection`. El destructor[18] lleva a cabo la

[18]Teóricamente el destructor nunca será invocado si no se llamó con anterioridad al constructor. Por tanto, podemos usar el método `Free` de manera segura, liberando la memoria asignada para el objeto. No obstante, también podríamos haber introducido una comprobación previa, en la forma `if Assigned(refInstance) then proceder`.

acción complementaria, destruyendo el objeto y asignando el valor `nil` al atributo `refInstance`.

La llamada que se hace desde el constructor de clase a `Create` no debemos confundirla con una llamada a sí mismo, lo cual provocaría una llamada recursiva infinita. Lo que estamos haciendo es delegar la tarea de crear el nuevo objeto en el constructor de instancia. Como se muestra en el código siguiente, este constructor llamará al constructor heredado, agregando una verificación adicional para evitar la creación de más de un objeto de la clase `TCollection`.

```
...

constructor TCollection.Create;
begin
  inherited;
  if refInstance <> nil then
     raise Exception.Create(
      'No se pueden crear más objetos de esta clase');
end;
```

Listado B.60 Implementación del constructor de instancia

La excepción no debería tener lugar nunca, dado que este constructor solo puede ser llamado desde el constructor de clase, en cuyo cuerpo ya se ha comprobado el contenido del atributo `refInstance`.

Este es muy sencillo, ya que no tiene más que tomar el valor almacenado en el atributo `refInstance` y devolver, tanto si se ya se ha creado un objeto como si no.

```
...

class function TCollection.getInstance: TCollection;
begin
  result := refInstance;
end;
```

Listado B.61 Implementación del método `getInstance`

Una vez concluida la implementación de la clase, podríamos acceder a los datos de la colección desde cualquier punto del proyecto. Primero obtendríamos una referencia a la colección, llamando al método `getInstance` o leyendo la propiedad `Instance` de la clase `TCollection`. A continuación, usaríamos esta referencia para acceder a cualquier miembro de instancia. Por ejemplo:

```
...

var
  MyCollection: TCollection;

...

begin
  MyCollection := TCollection.getInstance;
...
```

Listado B.62 Uso de la clase `TCollection`

> **NOTA**
>
> *Puesto que la clase `TCollection` está pensada para un uso final, no para ser usada como base de otras clases, podría ser marcada como clase final incluyendo la palabra reservada `sealed` tras la palabra `class`, en la cabecera de la clase. Esto impediría la creación de nuevas clases derivadas a partir de `TCollection`.*

B.7 Métodos anónimos

Hasta el momento, todos los procedimientos y funciones que hemos definido en los ejemplos previos han tenido asociado un nombre. Este nombre identifica de manera única cada método, de forma que podamos llamarlos posteriormente siempre que precisemos su funcionalidad. Podría decirse que los métodos, tal y

como los conocemos, representan el camino natural para agrupar sentencias a fin de poder ejecutarlas cada vez que lo necesitemos.

Delphi permite obtener la dirección de cualquier método, guardarla y pasarla como argumento a otros métodos. En cuanto al tipo de dato que le corresponde, esa información es un puntero (o una referencia si lo prefieres) a un método. No obstante, este tipo de dato solo es útil para apuntar a un método que continúa teniendo un nombre, una definición y una implementación.

Los métodos anónimos, por el contrario, son fragmentos de código que no tienen asociado un nombre. En su lugar son directamente asignados a una variable, o entregados como parámetro a un método, o incluso devueltos como resultado de la llamada a una función. Los métodos anónimos tienen múltiples aplicaciones.

B.7.1 Escenario de uso de un método anónimo

Un escenario típico en el que puede resultar de utilidad un método anónimo lo encontramos en la implementación de algoritmos de ordenación genéricos, cuya finalidad es ofrecer al usuario la libertad de elegir entre varias funciones de comparación. De esta forma el mismo algoritmo nos permitiría ordenar los elementos de menor a mayor valor o bien de mayor a menor valor, por ejemplo.

Una función de esta clase es la que se muestra continuación:

```
1  procedure Sort(items: TStringDynArray;
2                 compareFunc: TFuncComparator);
3
4  var
5    index, position: Integer;
6    temp: String;
7  begin
8    for index := Low(items)+1 to High(items) do
9      begin
10       temp := items[index];
11       position := index - 1;
12
13       while ((position >= Low(items)) and
```

```
14              compareFunc(temp, items[position])) do
15        begin
16          items[position+1] := items[position];
17          dec(position);
18        end;
19
20       items[position+1] := temp;
21    end;
22 end;
```

Listado B.63 Función de ordenación típica

Este algoritmo de ordenación es conocido popularmente como ordenación por inserción, siendo bastante simple y también eficiente cuando se utiliza con colecciones de datos pequeñas. El primer parámetro recibido por la función es de tipo `TStringDynArray`. Es el nombre que se asigna en el módulo `System.Types` al tipo `array of String`. Por tanto se trata de una matriz dinámica de cadenas de caracteres.

B.7.2 Definición de tipo de un método anónimo

Sin embargo, la parte más importante del anterior código es el segundo parámetro, llamado `compareFunc`. Como puede observarse en el cuerpo del procedimiento, dicho parámetro es usado como si fuese una función, en el interior del bucle `while`. En ese momento se le entregan dos argumentos, esperando obtener como resultado un valor booleano que nos indique si están en el orden correcto o no.

El tipo de dato `TFuncComparator` ha de ser definido con anterioridad a su uso, como cualquier otro tipo. Los detalles a tener en cuenta para definirlo son los siguientes: saber si apuntará a un procedimiento o a una función, la lista completa de parámetros de entrada, y el tipo del valor de retorno si fuese una función. Para este ejemplo la definición sería la siguiente:

```
1 type
2 TFuncComparator = reference to function(l1,l2: String): ↙
      ↳ Boolean;
```

Listado B.64 Definición de tipo para el método anónimo

B.7.3 Cómo pasar métodos anónimos como parámetros

Para utilizar el anterior procedimiento de ordenación tenemos que entregarle una matriz de cadenas y una función de comparación. Es lo que se hace en el siguiente código de ejemplo, en forma de método anónimo en lugar de declarar una función con nombre y entregarla como argumento:

```
...
var
  months: TStringDynArray;

...

begin
  months := SplitString('Ene,Feb,Mar,Abr,May,Jun,' +
                        'Jul,Ago,Sep,Oct,Nov,Dic', ',');

  Sort(months, function(l1, l2: String): Boolean
    begin
      if l1 < l2 then
        result := true
      else
        result := false;
    end);
```

Listado B.65 Método anónimo como parámetro en una llamada

Como puede apreciarse, el segundo parámetro facilitado a la función `Sort` comienza con la palabra reservada `function`, indicando por tanto que se trata de un método anónimo. Se especifica cuáles son los parámetros y el valor de retorno, pero no se coloca un punto y coma al final de la cabecera. Las sentencias forman un bloque, entre las palabras clave `begin` y `end` como es habitual, pero de nuevo sin colocar un punto y coma al final del bloque. El símbolo que señala el final del método es el cierre de paréntesis, justo tras la palabra `end`.

Tras la invocación al método `Sort`, los elementos existentes en la matriz que se entregó como primer parámetro ya estarán ordenados. El contenido

inicial de esta matriz se obtiene dividiendo una cadena mediante la función `SplitString`, cuyo segundo argumento establece el carácter que separa a cada uno de los elementos.

Dado que el método anónimo devuelve true si el primer parámetro se considera menor que el segundo, comunicando así que el orden en que están actualmente es correcto, el resultado será una ordenación de menor a mayor. Lo interesante es que el mismo algoritmo también serviría para ordenar a la inversa. Todo lo que tenemos que hacer es entregar como función anónima una en la que se sustituya el operador < por el operador >. De forma análoga, podría escribirse una función de comparación que comparase las cadenas sin distinguir entre mayúsculas y minúsculas. Por tanto la función que actúa como algoritmo de ordenación no sufriría cambio alguno para ajustarse a estas variadas necesidades.

B.8 Tipos genéricos

A la hora de planificar la definición de una nueva clase, generalmente se hacen algunas suposiciones en cuanto a los tipos de los datos con que deberá trabajar. Por ejemplo, en la sección anterior se ha asumido que el método de ordenación recibirá una matriz de elementos que son cadenas de caracteres. Sin embargo, podríamos necesitar ordenar una matriz que contuviese elementos numéricos, o quizá fechas.

El algoritmo de ordenación en sí sería exactamente el mismo, la única diferencia sería el tipo de dato asociado a los elementos contenidos en la matriz, que es el mismo tipo que se da a la variable local declarada dentro de la función, así como el mismo tipo asignado a los argumentos del método anónimo.

Delphi nos permite parametrizar las definiciones de clases y métodos en base a uno o más tipos de datos. El resultado será una clase (o método) genérica, dado que pueden ser aplicada a virtualmente cualquier tipo de dato[19]. La clase específica será generada una vez que se facilite un tipo concreto. En este sentido,

[19]Es posible establecer restricciones en cuanto a los tipos de datos específicos que pueden ser utilizados con un tipo genérico. No obstante, por el momento supondremos que puede emplearse cualquier tipo de datos.

los tipos genéricos de Delphi son muy parecidos a las plantillas de C++ o los tipos genéricos de Java.

Como ocurre con cualquier clase, el primer paso para poder usar un tipo genérico será aportar su definición. Una vez definido, será posible instanciarlo con un tipo de dato específico, dando valor a todos los parámetros usados como tipos en la definición original. Finalmente, las clases obtenidas a raíz de la instanciación se usarán como cualesquiera otras.

B.8.1 Cómo definir un tipo genérico

La característica principal de la definición de un tipo genérico es la inclusión de uno o más parámetros, delimitados entre los símbolos < y >, tras el nombre del tipo. Cada uno de esos parámetros es un identificador que posteriormente, durante la instanciación del tipo genérico, será sustituido por un tipo de dato específico. La sintaxis básica a usar es la mostrada a continuación:

```
1 NombreTipo<Tipo1, ..., TipoN> = class
2
3   // Los parámetros TipoI se usarán como tipos de datos
4
5   // dentro de la definición de la clase
6 end;
```

Listado B.66 Sintaxis de definición de tipos genéricos

Cada parámetro facilitado entre los delimitadores <> podrá después usarse para indicar el tipo de dato de los atributos y parámetros de métodos, así como servir como tipo base para definición de otros tipos de datos más complejos.

Supongamos que necesitamos utilizar el anterior algoritmo de ordenación con distintos tipos de datos, no únicamente con cadenas de caracteres. Fundamentalmente hay dos formas de afrontar este problema: creando un método de ordenación independiente para cada tipo de dato, usando la técnica de sobrecarga esos métodos podrían tener todos el mismo nombre, o bien definiendo un tipo genérico capaz de generar automáticamente los métodos específicos.

En el código de ejemplo mostrado a continuación se ha optado por el segundo de los enfoques citados:

```
...
type

TSort<DataType> = class
 type
   ArrayType = array of DataType;
   TFuncComparator = reference to
       function(l1,l2: DataType): Boolean;

   class procedure Sort(items: ArrayType;
                        compareFunc: TFuncComparator);
end;
...
```

Listado B.67 Definición de la función de comparación genérica

Como puede apreciarse, la clase genérica `TSort` necesitará un solo parámetro: `DataType`. Este es usado para definir un tipo de dato compuesto, al que hemos llamado `ArrayType`, y también como tipo de dato de los parámetros entregados a la función de comparación. El método de ordenación propiamente dicho se ha declarado como un procedimiento de clase, a fin de que pueda ser invocado directamente sin necesidad de crear antes un objeto de la clase `TSort`.

En cuanto a la implementación del método `Sort`, hemos de tener en cuenta que el tipo de los elementos de la matriz ya no será `String`, como en la versión original, sino `DataType`. En consecuencia, tenemos que cambiar el tipo en su cabecera, así como también la declaración de la variable temporal. El resto del código no experimentará ningún cambio, como puede verse a continuación:

```
class procedure TSort<DataType>.Sort(items: ArrayType;
                        compareFunc: TFuncComparator);
var
  index, position: Integer;
  temp: DataType;

```

```
7  begin
8    for index := Low(items)+1 to High(items) do
9      begin
10       temp := items[index];
11       position := index - 1;
12       while ((position >= Low(items)) and
13            compareFunc(temp, items[position])) do
14         begin
15           items[position+1] := items[position];
16           dec(position);
17         end;
18
19        items[position+1] := temp;
20     end;
21 end;
```

Listado B.68 Implementación del método `Sort` genérico

Fijémonos atentamente en la cabecera del método. Su nombre es precedido por el nombre de la clase a la que pertenece, así como la misma lista de parámetros que se indicó durante la definición. De esta forma lo que estamos dando al compilador es una plantilla de cómo ha de implementarse el método `Sort`. La plantilla será utilizada posteriormente para producir tantas versiones distintas del método como tipos de dato específicos se necesite tratar.

B.8.2 Instanciación y uso de tipos genéricos

Tras la definición, el tipo genérico ya está dispuesto para ser instanciado y utilizado con los tipos de dato específicos que pudiéramos necesitar. El proceso de instanciación generará una clase específica para cada tipo de dato concreto que especifiquemos.

Supongamos que estamos interesados en ordenar tanto cadenas de caracteres (`String`) como números enteros (`Integer`). Por tanto, necesitamos dos instancias distintas de la clase genérica `TSort`, tal y como se muestra en el código siguiente:

```
...
type
  TSortString = TSort<String>;
  TSortInteger = TSort<Integer>;
```

Listado B.69 Tipos concretos de la clase genérica

De este punto en adelante `TSortString` y `TSortInteger` serán dos clases diferentes, pero que ofrecen los mismos servicios fundamentales: un tipo de dato `ArrayType` y un tipo `TFuncComparator`, así como el método `Sort`. Estos se usarían como se muestra a continuación:

```
var
  months: TSortString.ArrayType;
  days: TSortInteger.ArrayType;

begin
  SetLength(days, 10);
  SetLength(months, 12);

  // TODO: Introducir elementos en las matrices

  // Ordenar la matriz de enteros
  TSortInteger.Sort(days,
   function(l1, l2: Integer): Boolean
    begin
      result := l1 < l2;
    end);

  // Ordenar la matriz de cadenas de caracteres
  TSortString.Sort(months,
   function(l1, l2: String): Boolean
    begin
      result := l1 < l2;
    end);
```

Listado B.70 Instanciación y uso de las funciones genéricas

Como puede apreciarse, en las dos llamadas que se hacen al método `Sort` se entrega como parámetro un método anónimo. Ambos son casi iguales, solamente difieren en el tipo de los parámetros de entrada. Tras la llamada al método `Sort` de la clase, las matrices quedarán ordenadas. Análogamente podríamos usar la clase genérica `TSort` para ordenar matrices de cualquier otro tipo, siempre que seamos capaces de facilitar un método de comparación adecuado.

B.9 Ayudas a la escritura de código

Si has ido poniendo en práctica los ejemplos propuestos anteriormente, escribiendo el código por ti mismo, probablemente ya hayas entrado en contacto con alguna de las múltiples herramientas de asistencia a la escritura de código con que cuenta Delphi.

Varios de estos mecanismos se ponen en marcha automáticamente. Por ejemplo, cada vez que se escribe un punto tras el nombre de un objeto el editor abre una lista mostrando sus miembros. Si se introduce una apertura de paréntesis tras el nombre de un método, sus parámetros aparecen en una pequeña ventana flotante. Estas son ayudas que se describieron en el apéndice previo, al tratar el entorno de Delphi. Existen, no obstante, otros mecanismos de ayuda no tan inmediatos, pero igualmente interesantes.

B.9.1 Plantillas de código

Esta funcionalidad se desencadena al detectar la introducción de ciertas secuencias de caracteres en el editor. Si escribimos `for` y pulsamos la barra espaciadora (o el tabulador), se introducirá en el editor un plantilla de código como la mostrada en la Figura B.8.

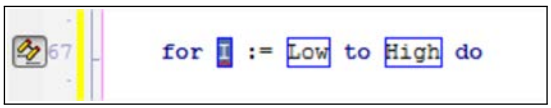


Figura B.8 PLANTILLA DE CÓDIGO PARA BUCLE `FOR`.

Una vez generada la plantilla de código, el puntero de texto se sitúa en el primer recuadro, esperando a que se introduzca el nombre de la variable. Pulsando el tabulador otra vez, el puntero se moverá al siguiente recuadro, facilitando la introducción de los límites inferior y superior del bucle. Este mecanismo está diseñado para ahorrarnos tiempo, ofreciendo un procedimiento abreviado para escribir algunas estructuras de uso común.

Si necesitáramos un bucle para recorrer los elementos de una matriz, en lugar de un bucle `for` clásico, tendríamos que escribir `forin` en lugar de `for`, lanzando así otra de las múltiples plantillas de código predefinidas.

Para ver qué plantillas predefinidas ofrece Delphi, elige la opción **TEMPLATES** del menú **VIEW**. Se mostrará una ventana como la de la Figura B.9, conteniendo una lista con todas las plantillas existentes. Los botones que hay en la parte superior permiten crear nuevas plantillas, así como eliminar y modificar las ya existentes.

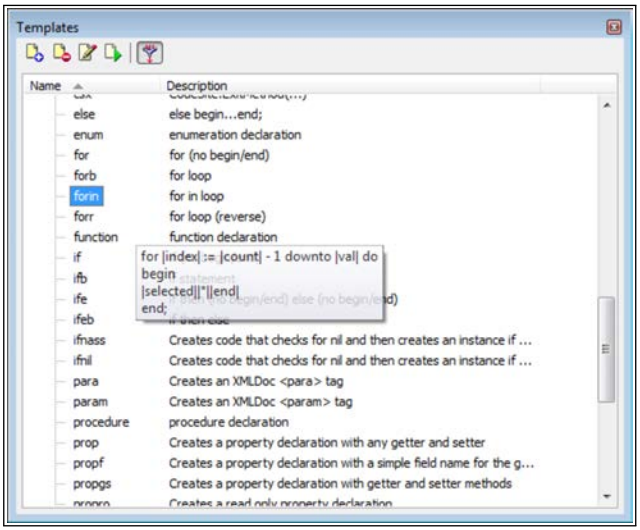


Figura B.9 LISTA CON LAS PLANTILLAS DE CÓDIGO PREDEFINIDAS

Además de cuando se utilizan plantillas, Delphi también puede generar código automáticamente en otras situaciones. Por ejemplo, cuando se inicia la definición de un nuevo tipo (como puede ser una clase). El editor agrega la palabra reser-

vada `end` al final por nosotros. Lo mismo ocurre cuando se abre un bloque de código, escribiendo la palabra clave `begin`, y se pulsa la tecla INTRO.

Otra ayuda útil de este tipo es el autocompletado de clases. Esta funcionalidad consiste en la generación automática de un esqueleto para todos los miembros de una clase que precisan de implementación. Prueba a definir un nueva clase con dos propiedades públicas de diferente tipo, como la mostrada a continuación:

```
1  TMyClass = class
2    public
3      property F1 : Integer;
4      property F2 : String;
5  end;
```

Listado B.71 Clase básica con dos propiedades

Tras colocar el puntero de texto en cualquier punto dentro del código de la clase, pulsa la combinación de teclas CTRL+MAYÚS+C para invocar la función de autocompletado. La definición de la clase se actualizará y quedará así:

```
1  TMyClass = class
2    private
3      FF2: String;
4      FF1: Integer;
5
6      procedure SetF1(const Value: Integer);
7      procedure SetF2(const Value: String);
8
9    public
10     property F1 : Integer read FF1 write SetF1;
11     property F2 : String read FF2 write SetF2;
12 end;
```

Listado B.72 Nueva definición de la clase tras completarla

Además, también se habrá añadido la implementación de los métodos `SetF1` y `SetF2` a la sección adecuada del módulo de código. Esa implementación

sencillamente asignara el parámetro recibido al atributo correspondiente, por lo que cualquier comprobación de validez de los parámetros habremos de añadirla personalmente.

B.9.2 Dar formato al código

A pesar de todas las ayudas que nos ofrece Delphi para la edición de código, llegando incluso a generarlo automáticamente en algunos casos, lo cierto es que la mayor parte del código de nuestro programa tendremos que escribirlo manualmente. Al hacerlo, no siempre es fácil mantener un estilo consistente y homogéneo. Más allá de aspectos estéticos, la aplicación de un formato adecuado al código también hace más cómoda su posterior revisión.

Es posible dar formato automáticamente al código que haya en el editor con la opción **EDIT—FORMAT SOURCE**. Si tenemos varios módulos de código en el proyecto, la opción **FORMAT PROJECT SOURCES** del menú **PROJECT** los procesará todos. El formato automático aplicado sigue la configuración establecida en la página **FORMATTER** de la ventana **OPTIONS** (véase la Figura B.10).

B.9.3 Navegar por el código

Al trabajar en proyectos de un cierto tamaño, con docenas de módulos de código conteniendo miles de líneas de código, localizar la sentencia exacta que nos interesa podría llegar a ser difícil. Por esta razón Delphi cuenta con varias opciones de utilidad en estas situaciones.

La herramienta más inmediata es la ventana **STRUCTURE**. En ella se muestra la estructura del módulo actualmente abierto en el editor, enumerando sus clases, tipos de datos y miembros. Un simple clic en uno de esos elemento desplazará el editor hasta el punto correspondiente a su declaración o implementación, dependiendo del tipo de elemento de que se trate.

Si lo que buscamos no se encuentra en el módulo actual, la ventana **STRUCTURE** no será de utilidad. En este caso podemos recurrir a una de las opciones del menú **SEARCH**, localizando una clase específica, buscando en todos los módulos del proyecto, encontrando todas las referencias a un símbolo, etc.

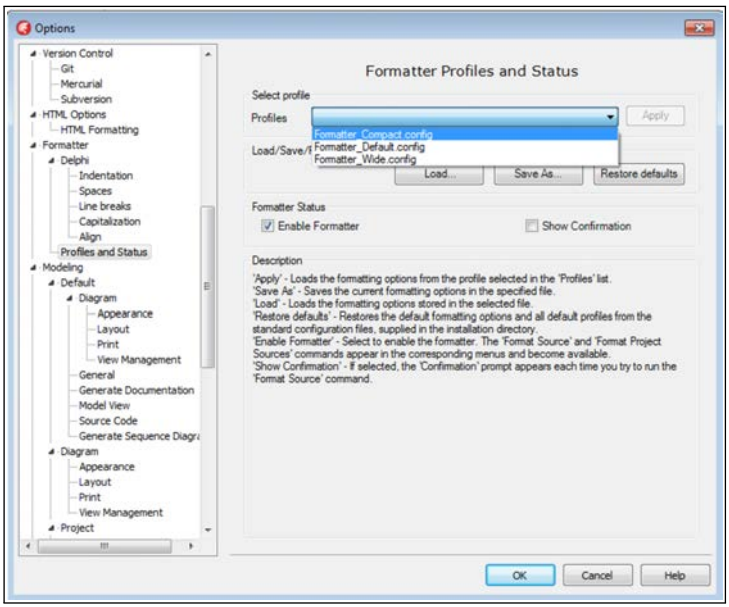


Figura B.10 SELECCIÓN DE UN PERFIL PARA DAR FORMATO AL CÓDIGO

Un atajo interesante del editor es el que nos permite acceder a la declaración o implementación de cualquier símbolo, sencillamente haciendo clic sobre su nombre mientras mantenemos pulsada la tecla **CTRL**, como si fuera un hipervínculo. Se abrirá el módulo en esté declarado el símbolo y se colocará el cursor en el punto correspondiente.

B.9.4 Refactorización del código

El menú **REFACTOR** contiene las opciones de refactorización de Delphi. La refactorización es una técnica que tiene por objetivo hacer más fácil la mejora continuada del código a medida que se trabaja en él.

Un ejemplo típico de refactorización es el siguiente: estamos escribiendo un bloque de sentencias para completar cualquier tarea, y más adelante nos damos cuenta de que necesitamos la misma acción en otro punto del módulo. En lu-

gar de recurrir a copiar y pegar el código, seleccionaríamos el bloque original de sentencia y elegiríamos la opción **REFACTOR—EXTRACT METHOD**. De esta forma se generaría automáticamente un método y se sustituiría el bloque de código original por una llamada a dicho método.

Algunas opciones de refactorización son fáciles de usar, como el renombrado de todas las apariciones de un símbolo, pero otras llevan a cabo tareas más elaboradas que pueden ahorrarnos mucho tiempo.

B.9.5 Gestión de versiones del código

A pesar de que el uso de un sistema de control de versiones de código fuente es siempre recomendable en cualquier proyecto software, cuando en dicho proyecto trabaja una sola persona esa recomendación raramente es satisfecha.

Delphi puede mantener las últimas versiones de cada módulo de código existente en un proyecto, ofreciendo opciones para comprobarlas y para restaurar versiones previas. Podemos abrir el historial de versiones del módulo actual mediante la pestaña **HISTORY** (en la parte inferior del editor).

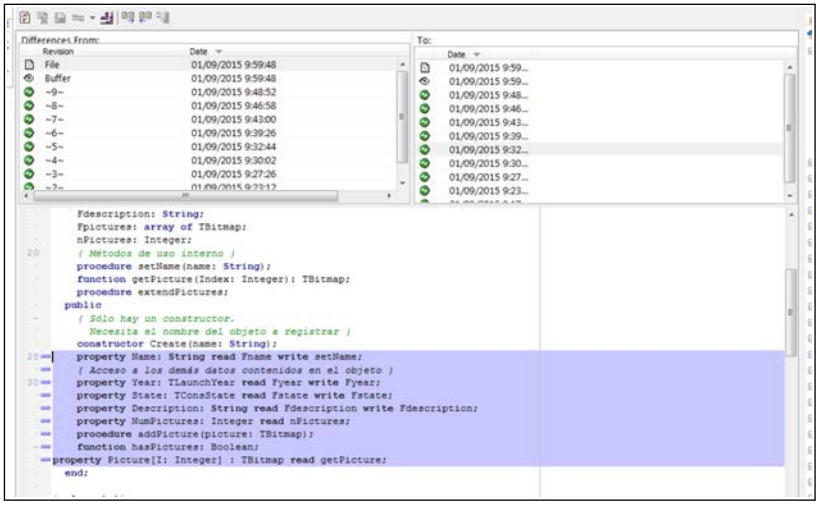


Figura B.11 PODEMOS COMPARAR VERSIONES Y RESTAURARLAS

Esta página muestra una lista de versiones en la parte superior. Podemos elegir dos versiones a fin de compararlas, mostrando las diferencias en el editor (véase la Figura B.11). Las líneas en las que hay diferencias se resaltan visualmente.

Aparte de la lista de versiones disponibles del módulo actual, inicialmente mostradas en orden cronológico inverso (desde la más reciente a la más antigua), en la parte superior encontramos una barra de herramientas con varias opciones. Dichos botones nos permiten comparar los módulos[20], añadir comentarios a cada versión, restaurar una versión anterior, etc.

NOTA

En el siguiente apéndice se describe la integración de Delphi con Git, la que probablemente es la herramienta más usada en la actualidad para gestionar versiones de proyectos.

B.10 Resumen

En este apéndice se ha añadido uno más de los pilares necesarios para nuestro conocimiento de Delphi. Una vez que hemos aprendido la bases del lenguaje de programación, aspecto que complementa el aprendizaje del IDE al que se dedicó el apéndice previo, estamos en disposición de leer los capítulos de este libro y poner en práctica los ejercicios propuestos sin ningún problema. A medida que lo hagamos iremos conociendo una buena parte de la biblioteca de componentes FMX y, sobre todo, aquellos elementos relacionados específicamente con el tratamiento de bases de datos.

[20]El motor encargado de llevar a cabo las comparaciones entre los módulos de código es una versión limitada del software Beyond Compare. La versión completa se incluye en algunas ediciones de Delphi. Puedes acceder a ella buscando Beyond Compare con IDE Insight. Si deseas la edición completa, contacta con tu comercial habitual en Danysoft .

APÉNDICE C

INTEGRACIÓN DE DELPHI CON GIT

Al trabajar en cualquier proyecto software deberíamos pensar no únicamente en la fase de desarrollo inicial, sino también en el posterior mantenimiento a medio o largo plazo, tarea en la que podemos intervenir una única persona o varias, dependiendo de la magnitud del proyecto y de su tiempo de vida. Por ello es importante contar desde el inicio con un **sistema de control de versiones** del código fuente.

Delphi incorpora un motor de control de versiones propio, gracias al cual podemos comprobar en cualquier momento el historial de cada módulo de código de un proyecto y examinar los cambios que se han ido realizando sobre él. No obstante su funcionalidad es limitada, de ahí que habitualmente se recurra a un software externo en el que confiar esta importante misión.

Aunque hay disponibles múltiples opciones para controlar el código fuente, tanto centralizadas como distribuidas y tanto comerciales como abiertas, la alternativa que ha ganado mayor terreno en los últimos años en este campo es **Git** (`http://git-scm.com`). Este es un sistema distribuido de control de código fuente. Algunas de sus características más destacables son:

- Cada desarrollador tiene su propia versión local del proyecto (un *repositorio* local), desde el que puede tanto obtener los cambios realizados por los demás programadores como aportar su propio trabajo, subiéndolo a un repositorio compartido.
- Varias personas pueden trabajar sobre el mismo módulo y Git se encargará de fusionar (operación *merge*) las aportaciones de cada una, obteniendo una versión unificada del código.
- Cada integrante del proyecto puede comprobar el estado de su repositorio local respecto a los demás, obtener los cambios de terceros, volver atrás en el historial de los módulos, notificar fallos y responder a notificaciones de terceros.
- Es posible crear *ramas* a partir de la versión principal del proyecto, desarrollando en paralelo dos o más versiones de un mismo proyecto a fin de que nuevas características no afecten al mantenimiento de la rama principal. Esas ramas pueden fusionarse en caso de que fuese necesario.

NOTA

En este capítulo se describirá cómo instalar y realizar algunas de las operaciones básicas con Git, centrándonos sobre todo en su integración con Delphi. Si quieres conocer más sobre Git puedes recurrir a la documentación del sitio oficial o a alguno de los múltiples libros que existen sobre este software.

C.1 Instalación de Git

Aunque Git nació en GNU/Linux actualmente está disponible para múltiples sistemas operativos, entre ellos Windows que es el que nos interesa, ya que el entorno de desarrollo de Delphi funciona sobre dicho sistema. En `http: //git-scm.com` encontraremos un enlace directo (véase la Figura C.1) para descargar la última versión disponible de Git para Windows.

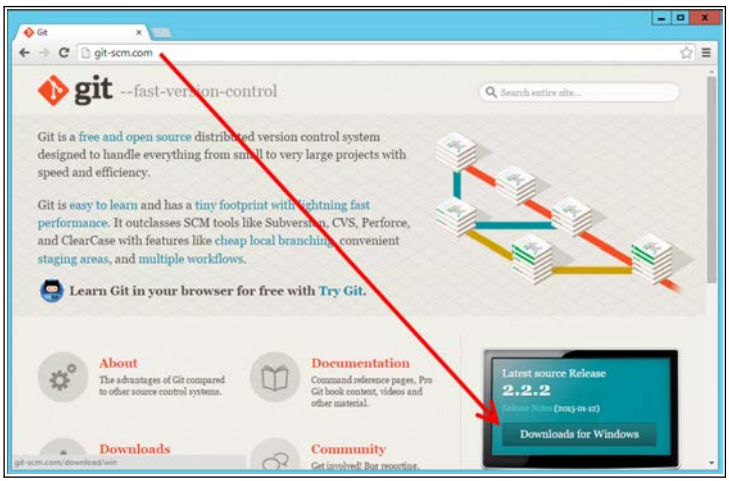


Figura C.1 PÁGINA DESDE LA QUE DESCARGAREMOS GIT

Una vez completada la descarga ejecutaremos el instalador que, como es habitual, constará de múltiples páginas en las que podremos ajustar distintos parámetros de configuración, entre ellos el directorio donde se instalará Git y la forma en que se integrará con el **EXPLORADOR DE ARCHIVOS** de Windows.

En uno de los pasos de configuración, el representado en la Figura C.2, tendremos que elegir entre tres alternativas diferentes a la hora de usar la línea de comandos de Git. La opción por defecto no precisa ningún ajuste del sistema, pero a cambio solo nos permitirá usar Git desde una línea de comandos Bash. La segunda opción habilita el uso de Git desde la ventana **SÍMBOLO DEL SISTEMA**

de Windows, pero para ello es necesario actualizar la variable `PATH`, algo que raramente representa un problema[1].

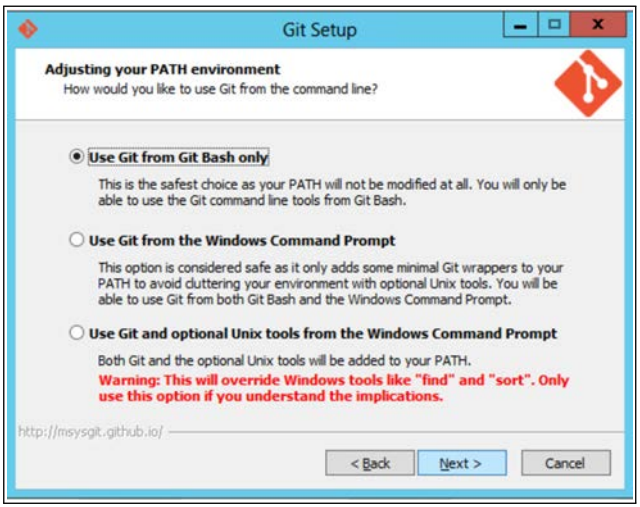


Figura C.2 CONFIGURACIÓN DE LA LÍNEA DE COMANDOS DE GIT

Los siguientes pasos del asistente se encargan de configurar otros aspectos del funcionamiento de Git, por ejemplo la conversión entre LF y CR/LF en los archivos de texto. Finalmente se pondrá en marcha el proceso de copia de archivos y configuración, finalizado el cual ya tendremos a nuestra disposición Git.

C.1.1 Uso de Git en Windows

Completada la instalación, en Windows tenemos básicamente dos opciones de uso de Git: una línea de comandos o una interfaz de usuario rudimentaria. Encontraremos ambas simplemente escribiendo `git` en la pantalla de inicio de Windows (véase la Figura C.3). Aunque desde la GUI es posible crear reposito-

[1]Bash es el intérprete de comandos más popular de Linux. Si estamos habituados a utilizarlo es la opción más recomendable, ya que nos permitirá usar las mismas órdenes que utilizaríamos en Linux.

rios, clonarlos y gestionar la mayor parte de las operaciones sobre los mismos, la línea de comandos resulta mucho más flexible. Además será la misma forma en que trabajaríamos con Git en otros sistemas, razón por la que es el método que recomendamos.

Figura C.3 ACCESO A LA GUI Y LÍNEA DE COMANDOS DE GIT

Al abrir la línea de comandos de Git nos encontraremos con una consola de Bash o bien con el SÍMBOLO DEL SISTEMA de Windows, dependiendo de la opci ón que hubiésemos elegido durante la instalación. En cualquier caso los comandos de Git serán los mismos, pero no así el resto de órdenes que usar íamos para, por ejemplo, obtener el contenido de un directorio, copiar un archivo o examinar su contenido, etc.

C.1.2 Configuraci ón de Git

Lo primero que hemos de hacer tras completar la instalación de Git es configurar nuestro nombre de usuario y dirección de correo electrónico, datos indispensables para identificar de manera apropiada las aportaciones que hagamos a los proyectos en que estemos trabajando. Para ello, desde la línea de comandos de Git, usaremos el comando `git config` dos veces: una para establecer el nombre de usuario, con el parámetro `user.name`, y otra para el correo electrónico, mediante el parámetro `user.email`. En el Listado C.4 se muestran dichos comandos. Lógicamente el texto entre comillas tendremos que cambiarlo por nuestros datos reales.

```
1 git config --global user.name "nombreUsuario"
2
```

```
3 git config --global user.email "dirección@correo"
```

Listado C.1 Configuración inicial de Git

Tras ejecutar los dos anteriores comandos podemos comprobar cuál es la configuración actual de Git, escribiendo en la consola `git config --list` y pulsando **INTRO**, tal y como se muestra en la parte inferior de la Figura C.4. Entre otros parámetros deberíamos ver nuestro nombre de usuario y dirección de correo electrónico.

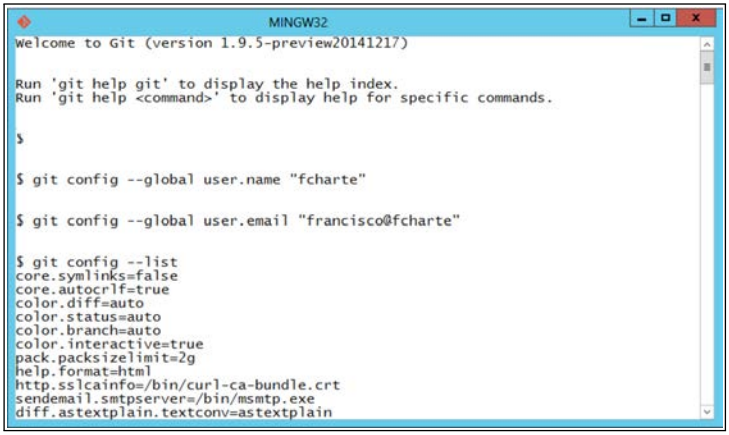


Figura C.4 CONFIGURACIÓN INICIAL DE GIT

NOTA

Si tenemos cuenta abierta en GitHub (`https://github.com`), podemos configurar nuestro Git local con el mismo nombre de usuario y direción de correo. Esto simplificará las operaciones de recepción y envío de cambios a repositorios alojados en la nube, al no tener que facilitar otros datos distintos.

C.2 Integración de Git con Delphi

Una vez que Git está instalado en nuestro sistema llega el momento de activar la integración con Delphi. Para ello abriremos el menú **TOOLS** y seleccionaremos la opción **OPTIONS**, abriendo en la ventana del mismo nombre la página **VERSION CONTROL—GIT**. Es la página que se muestra en la Figura C.5. En la parte superior tenemos un recuadro de texto en el que hemos de facilitar la ruta y nombre del ejecutable `git.exe`. Este se encontrará en el subdirectorio `cmd` de la carpeta en la que hubiésemos instalado Git.

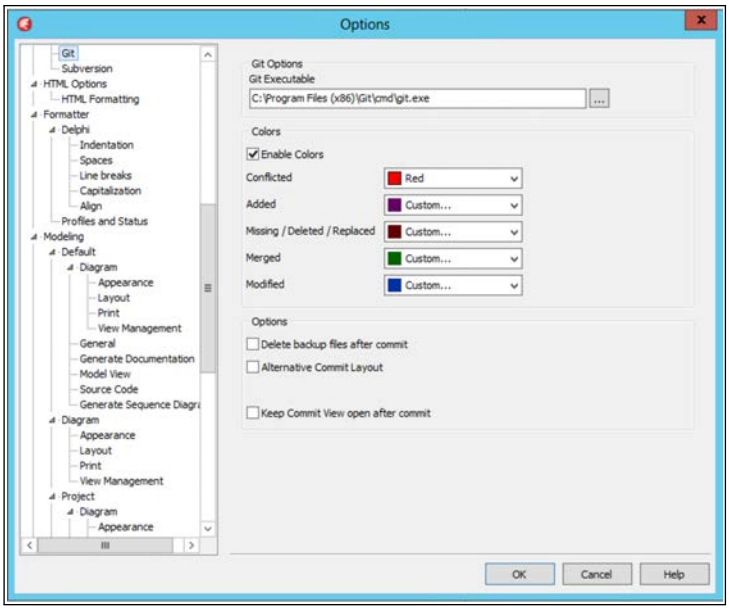


Figura C.5 CONFIGURACIÓN DE GIT EN EL ENTORNO DE DELPHI

Opcionalmente podemos configurar otros parámetros, como el borrado autom´atico de archivos de copia de seguridad tras cada *commit* o los colores que se usarán para representar los cambios. Terminaremos haciendo clic en el botón **OK** de dicha ventana. A partir de este momento ya podremos usar Git desde el entorno de Delphi, tal y como se explica en los apartados de la sección siguiente.

C.3 Operaciones comunes al trabajar con Git y Delphi

Asumiendo que hemos completado los pasos descritos en las secciones previas de este apéndice, y por lo tanto tenemos Git instalado e integrado con Delphi, veamos cuáles serían los procedimientos a seguir para trabajar con proyectos alojados en un repositorio Git.

> **NOTA**
>
> *Un repositorio Git es un directorio en el que existe una carpeta llamada* `.git`*. Esta actúa como almacén de toda la información necesaria para saber qué archivos están bajo control del sistema de control de código fuente, qué cambios ha sufrido cada archivo a lo largo del tiempo, etc. Podemos tener tantos repositorios como necesitemos, habitualmente uno independiente para cada proyecto.*

C.3.1 Clonación de un repositorio Git

En caso de que vayamos a trabajar en un proyecto ya existente, alojado en un repositorio remoto, el primer paso que habremos de dar es la creación de un *clon* de dicho repositorio en nuestra máquina. Desde la línea de comandos Git usaríamos la orden `git clone`, transfiriendo toda la estructura remota a nuestra máquina. Esta misma operación se efectúa desde Delphi con la opción **FILE—OPEN FROM VERSION CONTROL**. Esta da paso a un pequeño cuadro de diálogo en el que tendremos que elegir el software de control de versiones que usaremos. Elegimos la opción **GIT** y hacemos clic en el botón **OK**, dando paso a la ventana mostrada en la Figura C.6. Esta cuenta con dos apartados:

- **SOURCE**: Aquí hemos de introducir la ruta donde se encuentra el repositorio que queremos clonar. Normalmente se tratará de una ruta a un equipo remoto, ya sea en nuestra propia red local, por ejemplo si trabajamos con un grupo de personas y contamos con un servidor compartido, o bien a un servicio en la nube como puede ser GitHub.

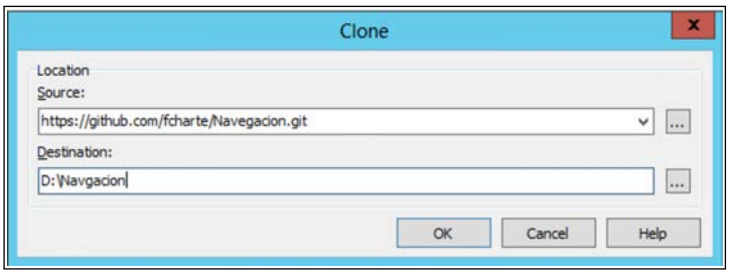


Figura C.6 CLONACIÓN DE UN REPOSITORIO REMOTO

- **DESTINATION**: Es la ruta en nuestro equipo donde se creará el repositorio local, sobre cuyos archivos trabajaremos desde Delphi. Lo habitual es seleccionar una carpeta vacía o crear una nueva para alojar el proyecto.

Una vez introducidos los dos parámetros, y pulsado el botón **OK**, se pondrá en marcha el proceso de clonación, durante el cual se transferirán los archivos remotos al directorio indicado de nuestro equipo y se inicializará un repositorio local. Durante este proceso se examinarán los archivos recibidos para dilucidar cuál es el proyecto o grupo de proyectos que hay que abrir en Delphi. Terminado el proceso nos encontraremos con un cuadro de diálogo como el que aparece en primer plano en la Figura C.7, ofreciéndonos la lista de proyectos encontrados. No tenemos más que seleccionar el que queramos abrir y hacer clic en **OK** una vez más.

Clonación de un repositorio GitHub

Si el repositorio que queremos clonar está alojado en GitHub lo primero que necesitaremos será su dirección, dato que hemos de facilitar en el recuadro de texto **SOURCE**. Para ello debemos localizar en GitHub el repositorio en que estemos interesados. A continuación buscaremos en su página el apartado **HTTPS CLONE URL**, compuesto de un recuadro de texto con el URL que nos interesa y, a su derecha, un botón para copiarlo al portapapeles (véase la Figura C.8). Finalmente, volveremos a Delphi y pegaremos la dirección del repositorio para clonarlo.

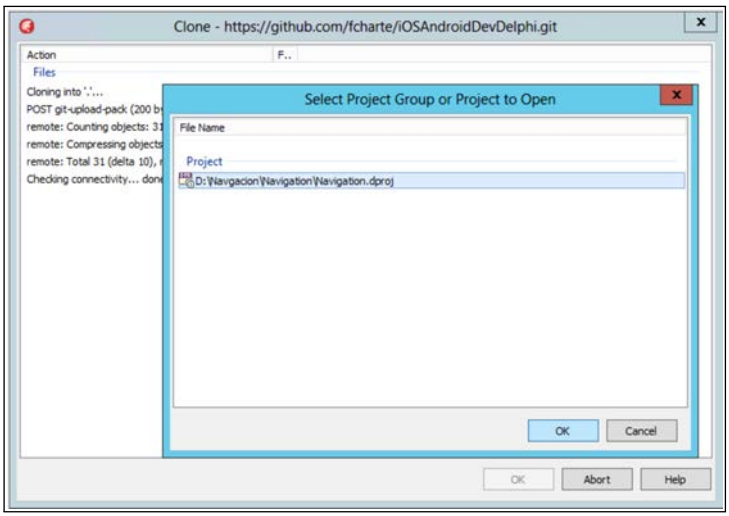


Figura C.7 PROCESO DE CLONACIÓN DEL REPOSITORIO REMOTO

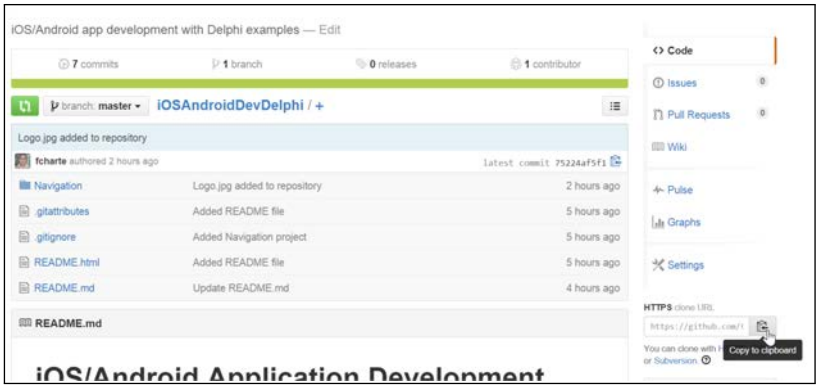


Figura C.8 OBTENER DIRECCIÓN DE CLONACIÓN DESDE GITHUB

NOTA

En GitHub podemos encontrar multitud de proyectos interesantes desarrollados con Delphi. Una búsqueda de `delphi` *desde la página principal de GitHub devuelve más de 1.500 repositorios listos para ser clonados en nuestro equipo. Los proyectos de ejemplo desarrollados en este libro también pueden encontrarse en GitHub, concretamente en* `https://github.com/fcharte/BddDelphi.git`

Clonación de un repositorio con autentificación

Al trabajar con repositorios internos a un grupo de trabajo o una empresa, alojados en equipos no públicos, normalmente nos bastará con conocer la ruta o dirección del repositorio para poder clonarlo. Lo mismo ocurre cuando el código está alojado en un repositorio público de GitHub. Si el repositorio es privado, no obstante, necesitaremos pasar por un proceso de autentificación para poder clonarlo. Este es un procedimiento no soportado actualmente[2] en Delphi.

En estos casos tendremos que recurrir a la línea de comandos de Git. Los pasos a seguir son los indicados a continuación:

- Usamos la opción **GIT BASH** para abrir la línea de comandos de Git.
- Cambiamos al directorio en el que queremos clonar el repositorio remoto, carpeta desde la que posteriormente abriríamos el proyecto en Delphi.
- Introducimos la orden `git clone` *`ruta`*, siendo *`ruta`* la dirección completa del repositorio a clonar.
- Git se pondrá en contacto con el repositorio remoto y, llegado el momento, nos pedirá que nos identifiquemos, facilitando un nombre de usuario y una contraseña. También sería posible autentificarse mediante una clave pública reconocida por el servidor que aloja el repositorio.

[2]Hasta Delphi XE7 únicamente se contempla el clonado de repositorios que no necesitan autentificación.

- Si la identificación es correcta se procederá a efectuar la clonación, como si la hubiésemos solicitado desde el entorno de Delphi siguiendo el procedimiento anteriormente descrito.

Con independencia de cómo hayamos efectuado la clonación, una vez terminada tendremos un repositorio local con la misma estructura y contenido. A partir de ese momento podemos comenzar a trabajar con el proyecto desde Delphi, como haríamos normalmente.

C.3.2 Inicializar un repositorio local

En la sección previa se asumía que el proyecto en el que íbamos a trabajar ya existía, por lo que procedíamos a clonarlo. No obstante, puede darse el caso de que seamos nosotros los que iniciemos el proyecto y, a partir de ahí, necesitemos inicializar un repositorio Git con él. Esto nos permitiría controlar los cambios en el código fuente a nivel personal y, si procede, permitir que otros usuarios clonen ese repositorio para trabajar en el mismo proyecto.

Por defecto Delphi no está preparado para inicializar un nuevo repositorio a partir de un proyecto, ya sea existente o recién creado. Tenemos básicamente dos opciones posibles: recurrir a la línea de comandos de Git cada vez que necesitemos inicializar un nuevo repositorio, usándola tal y como se describe en el siguiente apartado, o bien agregar a Delphi una opción para realizar dicha tarea, enfoque que se explicar en el apartado posterior.

Inicialización desde la línea de comandos de Git

Supongamos que acabamos de crear un nuevo proyecto en Delphi, proyecto que queremos poner bajo la supervisión de Git. Lo primero que hemos de hacer es crear una carpeta y guardarlo. La carpeta elegida será la que actúe como raíz del repositorio.

> **NOTA**
>
> *Git siempre opera sobre el sistema de archivos de nuestro equipo. Los cambios que tengamos en el editor y el diseñador de Delphi no son tenidos en cuenta hasta que no se escriben en los respectivos archivos.*

Acto seguido, a fin de que comunicar a Git que queremos que controle los cambios que se produzcan en el contenido de esa carpeta, abriríamos la consola y usaríamos el comando `git init`. Esto generaría el directorio `.git` anteriormente mencionado, con la información que necesita Git para realizar su trabajo.

En el entorno de Delphi el hecho de que el proyecto ya esté bajo supervisión del sistema de control de código fuente se aprecia en el menú contextual de los módulos en el **PROJECT MANAGER**, al agregarse un nuevo submenú con el título **GIT** (véase la Figura C.9) que no aparece en proyectos recién creados.

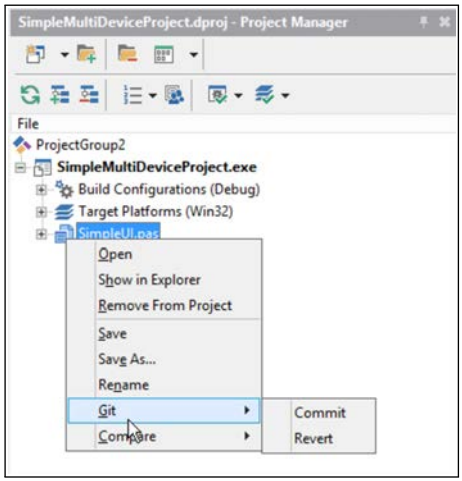


Figura C.9 MENÚ GIT AGREGADO AL MENÚ CONTEXTUAL DE LOS MÓDULOS

Agregar una opción de inicialización Git a Delphi

Delphi dispone de un mecanismo, asociado al menú **TOOLS**, que nos permite integrar herramientas externas en su entorno. Podemos utilizarlo para completar la tarea de inicialización de un repositorio sin necesidad de recurrir a la línea de comandos de Git. Los pasos a seguir para ello son los indicados a continuación:

1. Seleccionamos **TOOLS—CONFIGURE TOOLS** para abrir la ventana **TOOLS OPTIONS**.

2. Hacemos clic en el botón **ADD** para agregar una nueva herramienta a la lista, abriéndose la ventana **TOOL PROPERTIES**.

3. A continuación introducimos el título que queremos darle a la opción que se añadirá al menú, así como la ruta completa del programa que se quiere ejecutar, que en este caso es `git.exe`, y los parámetros que se le entregarán, que serán `init $PATH($PROJECT))`, tal y como se aprecia en la Figura C.10.

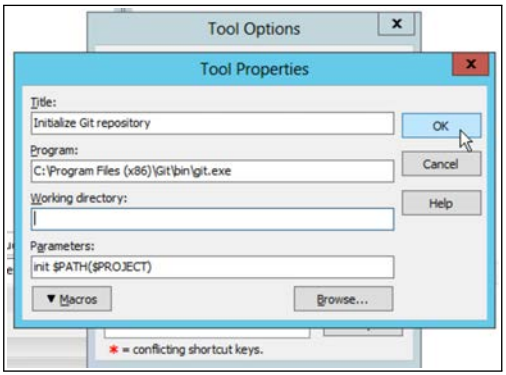


Figura C.10 CONFIGURAMOS LA NUEVA OPCIÓN PARA INVOCAR A `git init`

4. Hacemos clic en el botón **OK** de esta ventana y de la anterior, finalizando la definición de la nueva opción.

A partir de ahora, cada vez que creemos un nuevo proyecto y necesitemos controlar sus versiones con Git no tenemos más que abrir el menú **TOOLS** y elegir la nueva opción (véase la Figura C.11).

C.3.3 Comprobar el estado de nuestra versión local

Con independencia de que el repositorio correspondiente al proyecto en que trabajamos haya sido clonado de un repositorio remoto o creado en nuestro equipo,

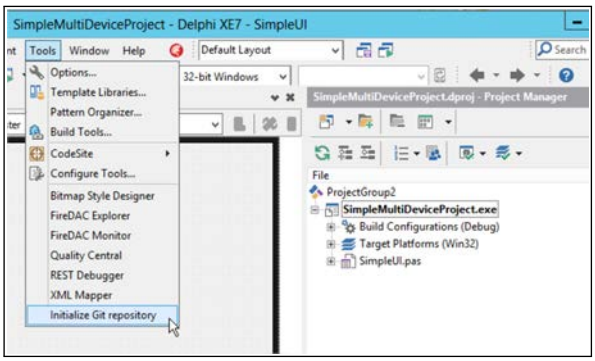


Figura C.11 USAMOS LA OPCIÓN PARA INICIALIZAR EL NUEVO REPOSITORIO LOCAL

las opciones que tenemos a nuestra disposición para verificar cuál es nuestro estado, confirmar cambios y enviarlos, solicitar cambios remotos, etc., serán fundamentalmente las mismas.

Para conocer cuál es el estado de nuestro proyecto, obteniendo una lista de archivos que han sido modificados o añadidos al repositorio, usaremos la opción **GIT—COMMIT—FROM PROJECT DIRECTORY** que podemos encontrar en el menú contextual del proyecto, en la ventana **PROJECT MANAGER** (véase la Figura C.12). Dicha opción abrirá la ventana **COMMIT** como una pestaña más, en el área del editor y diseñador de Delphi.

La ventana **COMMIT** mostrará en la parte superior una lista de los módulos que, estando previamente bajo control de Git, han sido modificados. Estos módulos tienen cambios pendientes de confirmar. No se muestran, por el contrario, los nuevos módulos que hayan podido añadirse. Para hacerlos visibles debemos marcar la opción **SHOW UNVERSIONED FILES**, situada en la parte inferior izquierda de la ventana (véase la Figura C.13). Esto nos permite agregar esos nuevos módulos al repositorio, sencillamente marcándolos[3] como lo están por defecto los modificados.

[3]Podemos marcar todos los archivos, tanto los que han sido modificados como los nuevos, mediante la opción **CHECK OR UNCHECK ALL** dispuesta en la parte inferior izquierda de la ventana.

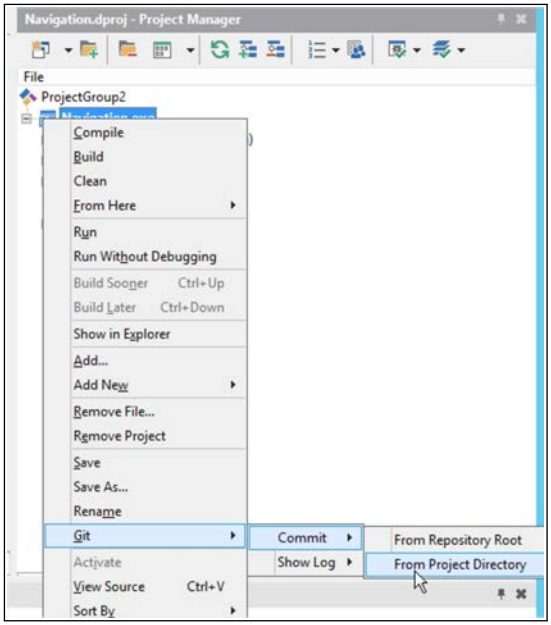


Figura C.12 OPCIONES DE GIT ASOCIADAS AL PROYECTO

Cada uno de los módulos enumerados en la ventana **COMMIT** cuenta con un menú contextual, tal y como se aprecia en la Figura C.14, en el que se nos ofrecen dos opciones: **DIFFERENCE** y **REVERT**.

La primera opción nos permite examinar las diferencias entre el estado actual del módulo y el estado en que estaba en cualquiera de las versiones previas alojadas en el repositorio. Como se aprecia en la Figura C.15, la parte superior de la ventana se divide en dos paneles con la misma lista de elementos, lo cual nos permite elegir cualquier pareja de estados y compararlos.

> **NOTA**
>
> *Podemos acceder a la ventana de diferencias de cualquier módulo que tengamos abierto en el editor mediante la pestaña* **HISTORY**.

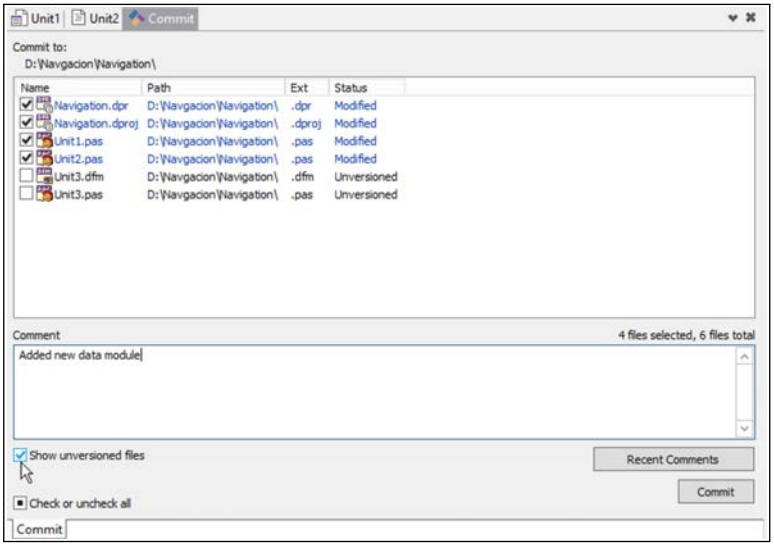


Figura C.13 LISTA DE MÓDULOS CON CAMBIOS Y AGREGADOS AL PROYECTO

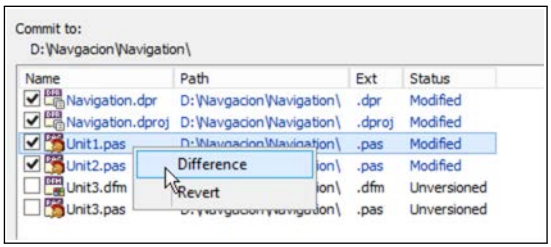


Figura C.14 MENÚ CONTEXTUAL ASOCIADO A CADA MÓDULO EN LA VENTANA COMMIT

Mediante la barra de herramientas que hay en la parte superior de la ventana podemos ir recorriendo las diferencias existentes, así como actualizar la información mostrada y ejecutar otras acciones sobre el módulo.

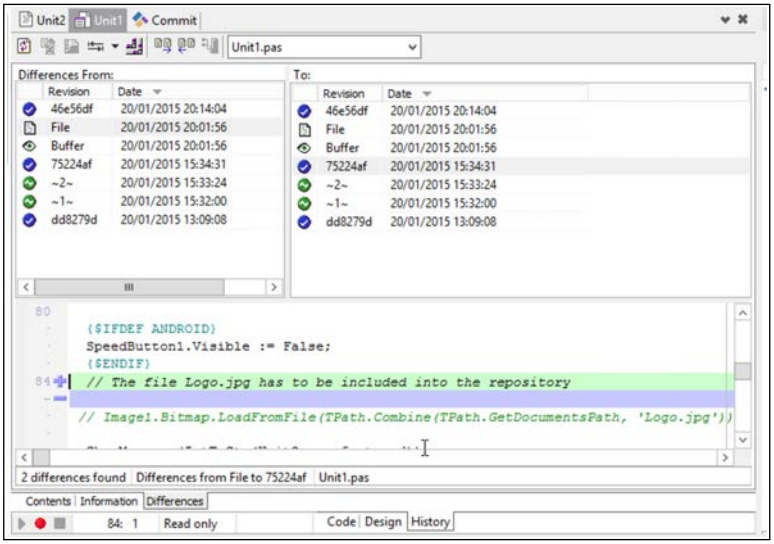


Figura C.15 REVISIÓN DE LOS CAMBIOS HECHOS A UN MÓDULO

C.3.4 Confirmar y revertir cambios en un módulo

La operación que ejecutaremos de forma más habitual al trabajar con un sistema de control de código fuente como Git es la confirmación de cambios realizados a uno o más módulos. Esta operación es conocida como *commit*, ya que el comando que hay que usar para ejecutarla es `git commit`.

Cada uno de los módulos, en la ventana **PROJECT MANAGER**, cuenta con su propio menú **GIT** desde el cual podemos abrir la ventana **COMMIT**. Una vez en ella marcaremos los módulos cuyos cambios queremos confirmar, introduciremos un comentario en el recuadro inferior aclarando qué se ha hecho y, finalmente, haremos clic en el botón **COMMIT**, confirmando los cambios. Git generará una firma única para identificar este *commit*, en el que incluirá el comentario y todos los cambios realizados. Esa información hará posible la revisión posterior de cambios y, si fuese preciso, la vuelta atrás a una versión previa.

La opción **REVERT**, que aparece tanto en el menú contextual de los módulos individuales en la ventana **PROJECT MANAGER** como en el de la lista de elementos de la ventana **COMMIT** antes descrita, tiene la finalidad de deshacer los cambios que afectan al módulo, devolviéndolo al estado en que estaba en el repositorio.

C.3.5 Enviar/Recibir cambios de un repositorio remoto

Al trabajar con Git debemos tener en cuenta que se trata de un sistema de control de versiones de código fuente distribuido, sin conexión permanente con un servidor. Esto implica que nuestros cambios, a pesar de que los hayamos confirmado como acaba de explicarse, únicamente afectarán a nuestro repositorio local y, por tanto, solo nosotros los veremos. Análogamente, tampoco nosotros veremos de forma automática en nuestro repositorio los cambios que realicen los demás integrantes del proyecto.

De forma periódica, y habitualmente siempre antes de comenzar a trabajar en un módulo, solicitaremos los últimos cambios que se hayan producido en el proyecto, actualizando nuestro repositorio local respecto al repositorio remoto. Con este fin abriremos la línea de comandos de Git e introduciremos la orden `git pull`[4]. Si existen cambios estos serán transferidos a nuestro repositorio local, en caso contrario únicamente se notificará que dicho repositorio está al día, como se aprecia en la parte inferior de la Figura C.16.

Habitualmente siempre que concluyamos la implementación de una modificación importante en el proyecto haremos un *commit*. Para que los demás miembros del equipo reciban esos cambios hay dos alternativas: que los demás conecten con nuestro equipo haciendo un `git pull` desde el suyo, de forma que nuestro repositorio haría las veces de centralizador del código, o bien que seamos nosotros los que transfiramos dichos cambios a un repositorio remoto. Esta última opción suele ser la más corriente.

[4] Al utilizar la consola de Git siempre se asume que nos encontramos en el directorio del proyecto o en uno de sus subdirectorios, de forma que los comandos pueden localizar en la carpeta `.git` la información que necesitan para poder realizar su trabajo.

Figura C.16 ACTUALIZAMOS NUESTRO REPOSITORIO LOCAL

Para enviar los cambios de nuestro repositorio local a uno remoto también recurriremos a la línea de comandos de Git, usando en este caso la orden `git push`.

En caso de que fuese necesaria autentificación, lo cual es habitual al operar sobre repositorios remotos para actualizar, se nos solicitará el nombre de usuario y contraseña (véase la parte superior de la Figura C.16). Acto seguido se transferirán los cambios al repositorio remoto y se actualizarán los archivos, combinando nuestras aportaciones con las del resto de miembros del equipo.

Si fuese necesaria nuestra intervención, por la aparición de algún tipo de conflicto, se nos informaría apropiadamente en la consola indicándonos cuál es el problema y cómo proceder.

NOTA

Al igual que hicimos anteriormente para la tarea de inicialización de un nuevo repositorio local, las órdenes `git pull` *y* `git push` *podrían agregarse al menú* **TOOLS** *de Delphi siempre y cuando no requieran autentificación.*

C.4 Configurar un repositorio Git

El comportamiento de Git al operar sobre un repositorio puede configurarse mediante los archivos `.gitignore` y `.gitattributes`. Estos los incluiremos en la carpeta raíz del proyecto, si lo hemos creado nosotros, o bien los recibiremos al clonar un repositorio remoto, en cuyo caso raramente los modificaremos.

Mediante el archivo `.gitignore` se establece qué archivos Git no debe tener en cuenta a la hora de mantener el repositorio. Es posible indicar módulos que tienen una cierta extensión, por ejemplo `*.dll` para que no se tengan en cuenta las bibliotecas de enlace dinámico, así como rutas que deben ser ignoradas, por ejemplo `_history/` para que Git ignore el contenido del directorio en el que Delphi mantiene su propio historial de cambios.

Lógicamente el contenido del archivo `.gitignore` depende del tipo de información que aloje un repositorio. En el caso de los proyectos Delphi no nos interesará que se controlen versiones de los archivos binarios generados al compilar la aplicación, módulos temporales para depuración, etc. A continuación se muestra cuál podría ser el contenido de `.gitignore` para nuestros repositorios de proyectos Delphi:

```
1  *.exe
2  *.dll
3  *.bpl
4  *.bpi
5  *.dcp
6  *.so
7  *.apk
8  *.drc
9  *.map
10 *.dres
11 *.rsm
12 *.tds
13 *.dcu
14 *.lib
15 *.o
16 *.vsr
17
18 *.cfg
```

```
*Resource.rc

*.local
*.identcache
*.projdata
*.tvsconfig
*.dsk

__history/
*.~*
Android/
iOSSimulator/

Thumbs.db
ehthumbs.db
Desktop.ini
$RECYCLE.BIN/
*.cab
*.msi
*.msm
*.msp
*.lnk
```

Listado C.2 Archivo `.gitignore` para proyectos Delphi

En cuanto al archivo de configuración `.gitattributes`, asigna a cada ruta facilitada atributos que determinan, por ejemplo, la forma en que se evaluarán las diferencias, la conversión de los finales de línea, el tratamiento de errores por presencia de espacios en blanco, etc. Podemos encontrar la documentación de todos esos atributos en `http://git-scm.com/docs/gitattributes`.

A continuación se muestra un ejemplo de archivo de configuración básico para proyectos Delphi, en el que se indica a Git qué módulos han de ser tratados como texto y cuáles como binarios, entre otros detalles:

```
# Tratamiento general de archivos de texto
* text=auto

# Módulos de proyecto y fuentes
```

```
5  *.pas            text            -whitespace
6  *.fmx            text            -whitespace
7  *.dfm            text            -whitespace
8  *.dpr            eol=crlf        -whitespace
9  *.dproj          eol=crlf        -whitespace
10
11 # Módulos binarios
12 *.dcr           -text
13 *.res           -text
```

Listado C.3 Archivo `.gitattributes` para proyectos Delphi

C.5 Resumen

En este apéndice se han descrito los procedimientos fundamentales para utilizar Git en proyectos Delphi, como herramienta de control de versiones del código fuente. Una vez instalado Git y configurada su integración en el entorno de Delphi tenemos a nuestra disposición opciones para clonar repositorios remotos, examinar los cambios que hemos llevado a cabo, confirmar dichos cambios, etc.

También se ha explicado cómo efectuar otras tareas para las que, por el momento, Delphi no ofrece opciones, tales como la inicialización de un nuevo repositorio o la transferencia de cambios entre nuestro repositorio local y uno remoto, en ambos sentidos. Estas son las operaciones que necesitaremos en el día a día, pero Git contempla muchas otras que no se han abordado como la creaciónn y fusiónn de ramas, gestión de conflictos, etc. Sobre todas ellas encontraremos información en la documentación oficial de Git.

APÉNDICE D

MIGRACIÓN DE APLICACIONES BDE A FIREDAC

El motor de acceso a datos BDE ha sido durante muchos años el método preferente para trabajar con bases de datos en aplicaciones Delphi. Tras la incorporación de dbExpress como solución alternativa multiplataforma y multi-RDBMS, en Delphi 6, una gran cantidad de desarrolladores prefirieron seguir utilizando BDE. Incluso muchos años después, con varias versiones incluyendo FireDAC como mecanismo preferente para el trabajo con bases de datos, BDE

sigue siendo la opción primera para muchos, a pesar de los muchos inconvenientes de BDE (véase enumeración en la descripción de BDE facilitada en la Sección 1.4.1, en la página 44) respecto a FireDAC.

BDE fue declarado obsoleto hace años y, de hecho, a partir de la versión XE7 del producto ni siquiera se incluye con la instalación de Delphi. Seguir contando con BDE, sin embargo, permite que muchas aplicaciones actualicen su interfaz de usuario, adecuándose a las nuevas versiones de Windows, o incorporando nuevas funcionalidades sin necesidad de un rediseño completo de los proyectos como el que forzaría la sustitución de BDE por otro mecanismo de acceso a datos.

El objetivo de este capítulo es ofrecer al lector unas nociones básicas sobre las principales diferencias existentes entre BDE y FireDAC, necesarias para comprender las implicaciones de una transición del primero al segundo, así como describir la herramienta que puede utilizar para simplificar el proceso de migración. Comenzaremos, no obstante, explicando cómo podemos seguir utilizando el BDE a partir de Delphi XE7.

D.1 Instalar BDE en XE7 y versiones posteriores

Con la versión XE7 de RAD Studio la empresa Embarcadero anunció que BDE había llegado a su etapa final, tras haberse declarado como tecnología obsoleta años antes. Por esta razón BDE ya no se incluía en la instalación del producto, como tampoco lo hacían los componentes asociados para acceder a sus servicios. Esto es algo que, lógicamente, afecta a todas las versiones posteriores.

Por el momento, sin embargo, existe una vía alternativa para seguir utilizando BDE en proyectos Delphi con XE7, XE8 y posiblemente versiones futuras. Esta consiste en instalar manualmente el BDE y la biblioteca de componentes BDE. Es una opción que solo tiene sentido para el mantenimiento de aplicaciones Win32 ya existentes en las que tengamos que introducir algún cambio.

El procedimiento para instalar BDE en las últimas versiones de Delphi es el descrito a continuación:

1. Lo primero que tenemos que hacer es entrar en nuestra cuenta de usuario registrado en `cc.embarcadero.com` y localizar la entrada correspondiente al instalador de BDE para nuestra versión de Delphi. Suponiendo que tuviésemos instalado en nuestro equipo Delphi XE8, la versión apropiada del instalador de BDE sería la mostrada en la Figura D.1. Hacemos clic en el enlace **DOWNLOAD** y guardamos el archivo.

Figura D.1 DESCARGA DEL INSTALADOR DE BDE PARA DELPHI XE8

2. Ejecutamos el archivo descargado para iniciar el instalador. Este no precisa información alguna, solo tendremos que hacer clic varias veces en el botón **NEXT**, aceptar la licencia de uso y completar la instalación (véase la Figura D.2).

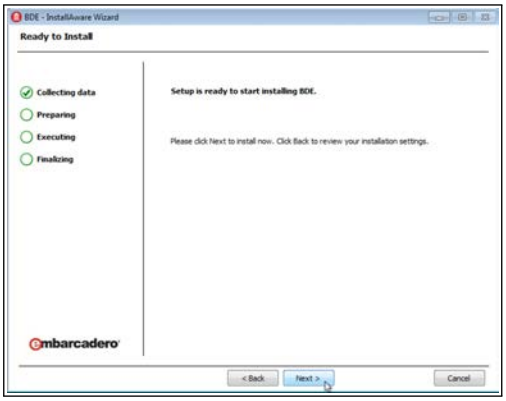


Figura D.2 INSTALACIÓN DE BDE EN UN EQUIPO CON DELPHI XE8

> **NOTA**
>
> *La instalación de BDE requiere que en el equipo esté ya instalada la versión de Delphi correspondiente. Si no es así el instalador generará un error y terminará.*

3. Una vez completada la instalación tendremos que agregar al IDE de Delphi el paquete de componentes de BDE. Para ello usaremos la opción **COMPONENT—INSTALL PACKAGES**, haremos clic en el botón **ADD** del cuadro de diálogo que se abre y localizaremos el paquete `dclbde210.bpl` en el subdirectorio `bin` del directorio donde esté instalado Delphi, como se muestra en la Figura D.3.

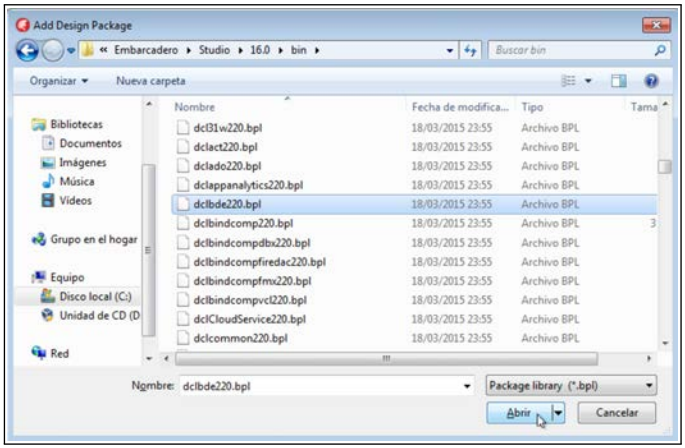


Figura D.3 LOCALIZAMOS EL PAQUETE AGREGADO POR EL INSTALADOR

4. Tras cerrar el cuadro de diálogo anterior volveremos al mostrado en la Figura D.4, en el que nos aseguramos de marcar la opción **EMBARCADERO BDE DB COMPONENTS** que se ha agregado a la lista tras abrir el módulo anterior. A continuación podemos cerrar esta ventana haciendo clic en **OK**.
5. Completados los pasos anteriores, en la Paleta de herramientas encontraremos el grupo de componentes **BDE** (véase la Figura D.5) listos para usar.

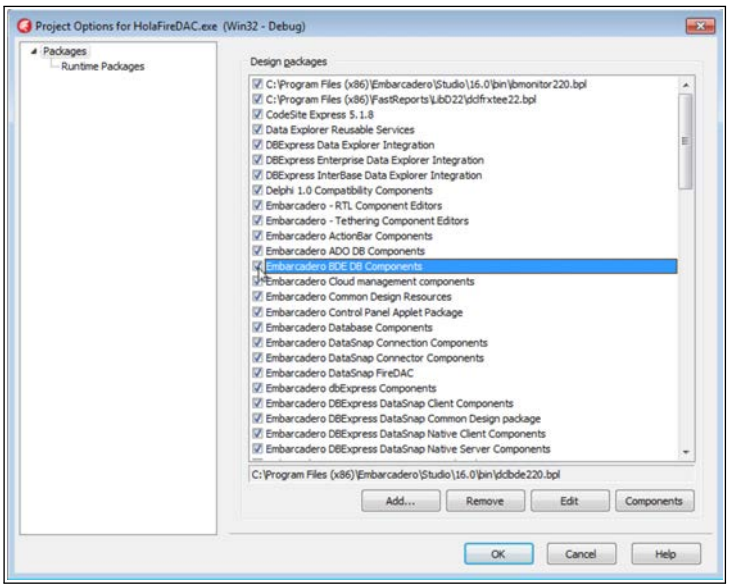


Figura D.4 ACTIVAMOS EL PAQUETE DE COMPONENTES BDE

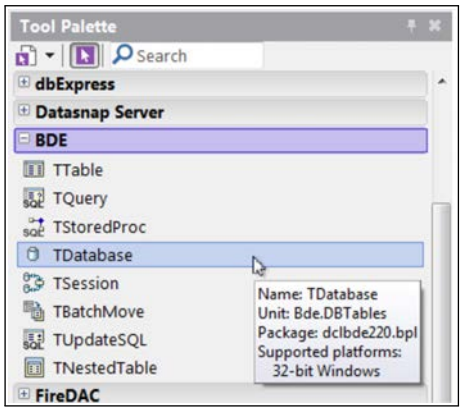


Figura D.5 COMPONENTES BDE EN LA PALETA DE HERRAMIENTAS

D.2 Diferencias entre BDE y FireDAC

Para poder convertir un proyecto en el que se utiliza BDE, con el objetivo de sustituirlo por FireDAC y así poder aprovechar todas las ventajas de este último mecanismo de acceso a datos, es necesario que conozcamos las diferencias existentes entre ambas soluciones. Este conocimiento nos permitirá determinar cuál es la vía de transición más adecuada a cada caso.

Atendiendo a la complejidad relativa que supondría la conversión, en los siguientes apartados se enumeran las diferencias más destacables y los problemas que plantearían.

D.2.1 Orígenes de datos

Tanto BDE como FireDAC cuentan con controladores específicos para cada uno de los orígenes de datos a los que permiten acceder. La naturaleza de los controladores es completamente distinta, ya que en el caso de FireDAC los podemos encontrar en la Paleta de herramientas en forma de componentes, incorporándose al propio código de la aplicación. La principal diferencia, no obstante, estriba en el conjunto de controladores ofrecidos por cada opción.

FireDAC cuenta con controladores para acceder a multitud de RDBMS, supliendo con suficiencia los SQL Links de BDE. Este, sin embargo, contaba con controladores nativos para trabajar con bases de datos locales como Paradox y dBase, inexistentes en FireDAC. Convertir una aplicación BDE que utiliza orígenes de datos no directamente accesibles desde FireDAC, como en los casos mencionados, conlleva elegir una de las dos siguientes alternativas:

- Recurrir a un controlador ODBC o dbExpress que permita acceder al origen de datos, usando en FireDAC el componente adecuado para conectar con ODBC, que sería `TFDPhysODBCDriverLink`, o con controladores dbExpress, que sería `TFDPhysTDBXDriverLink`. De esta forma estaríamos utilizando un controlador intermedio que, aparte de influir en el rendimiento, generalmente limitará la funcionalidad del controlador final que es posible utilizar. No obstante, esta solución permitiría seguir utilizando el mismo origen de datos existente.

- Trasladar la información alojada en el origen actual, por ejemplo una base de datos Paradox, a otro tipo de origen accesible desde FireDAC, como podría ser SQLite o InterBase. De esta forma la aplicación accedería a los datos con un controlador FireDAC nativo, sin intermediarios, obteniendo el mejor rendimiento y aprovechando toda la funcionalidad disponible.

En cualquiera de los dos casos, la conversión de una aplicación con esta casuística implica revisar en el código de la aplicación original todas aquellas sentencias en las que se empleen características específicas asociadas al antiguo controlador BDE, sustituyéndolas por aquellas del componente FireDAC que ofrezcan una funcionalidad equivalente. En el peor de los casos habría que eliminar funcionalidad que no sea fácilmente trasladable de un contexto a otro.

D.2.2 API de programación

Tanto BDE como FireDAC cuenta con una API de programación que podríamos calificar como de *bajo nivel*, frente a la de alto nivel ofrecida por los métodos de los distintos componentes. En el caso de BDE esa API es conocida como IDAPI, permitiendo operar directamente con los controladores y las funciones expuestas por estos.

Con FireDAC el API de bajo nivel es accesible mediante los métodos de la clase `TFDPhysDriver`, a la que se accede con la propiedad `DriverIntf` de la clase `TFDPhysDriverLink`, de la que descienden todos los componentes de la página **FIREDAC LINKS**.

Huelga decir que ambas interfaces de programación no tienen nada en común. Por tanto, cualquier proyecto en el que se use la API de BDE requerirá una revisión y conversión manual de todo el código afectado, adaptándolo para emplear la nueva API o bien rediseñar la funcionalidad original.

D.2.3 Componentes

Quizá la diferencia más patente para el desarrollador entre BDE y FireDAC sea el conjunto de componentes que tiene a su disposición. Algunos componentes de BDE tienen un equivalente muy similar en FireDAC, como es el caso de `TTable` o `TQuery` respecto a `TFDTable` y `TFDQuery`. Otros, por el contrario, tienen

nombres totalmente distintos, no son completamente equivalentes o, sencillamente, no tienen una correspondencia, representando una funcionalidad de BDE que no existe en FireDAC o viceversa.

A continuación se enumera, para cada uno de los componentes BDE de uso más habitual, cuál sería la vía de transición en el caso de FireDAC:

- `TDatabase`: Es el componente de BDE que establece los parámetros necesarios para conectar con una base de datos. El equivalente en FireDAC es el componente `TFDConnection`. Básicamente hay que sustituir uno por otro y adecuar los parámetros de conexión. El acceso desde el código a las propiedades y métodos del `TDatabase` habrá de ser revisado para adecuarlo a los servicios del `TFDConnection`. Por ejemplo: las llamadas al método `Execute` se convertirían en invocaciones al método `ExecSQL` del segundo.
- `TSession`: La funcionalidad de este componente BDE es equivalente a la que ofrece el componente `TFDManager` de FireDAC. Los componentes BDE que contienen una referencia a la sesión, en la propiedad `Session`, pasarían a emplear la propiedad `FDManager` de los componentes FireDAC equivalentes.
- `TTable`, `TQuery` y `TStoredProc`: Estos tres componentes serían sustituidos por `TFDTable`, `TFDQuery` y `TFDStoredProc`, conectándose con la base de datos mediante la propiedad `Connection` en lugar de `Database`.
- `TUpdateSQL`: Este componente, cuya finalidad es aportar los comandos SQL para actualizar resultados obtenidos de un procedimiento almacenado o una consulta no actualizable, tiene su equivalente en FireDAC en el componente `TFDUpdateSQL`.

Además de los componentes en sí, la conversión de un proyecto que usa BDE a FireDAC también requerirá la modificación de nombres de propiedades y métodos. La propiedad `UpdateMode` de los componentes BDE, por ejemplo, pasaría a ser `UpdateOptions.UpdateMode` en FireDAC. Muchos tipos de datos usados internamente por los componentes también han de convertirse. Las clases `TParam`, `TBlobStrem` y `TAutoIncField`, por ejemplo, pasan a denominarse `TFDParam`, `TFDBlobStream` y `TFDAutoIncField`, respectivamente.

Ciertas características de BDE sencillamente no existen en FireDAC, por lo que la única opción existente es su eliminación. Es el caso, por ejemplo, de la propiedad `WorkingDir` para establecer el directorio de trabajo.

> **NOTA**
>
> *En una sección posterior se explica cómo automatizar la eliminación y conversión de todas las referencias a componentes, propiedades y métodos con la herramienta de conversión incorporada en las últimas versiones de Delphi.*

D.2.4 Definición de conexiones

Tanto BDE como FireDAC permiten definir conexiones a las que se asigna un nombre y un conjunto de parámetros, información que se almacena en un archivo externo accesible desde las distintas herramientas del entorno. En el caso de BDE se denomina *alias*, mientras que en el FireDAC son sencillamente conexiones predefinidas con nombre.

Los alias de BDE se guardan en un archivo binario llamado `IDAPI.CFG`, común a todas las aplicaciones que usan este motor de acceso a datos. FireDAC cuenta con un archivo global de tipo `.ini` y también es posible utilizar archivos de conexiones, con el nombre `FDConnectionDefs.ini`, locales a cada proyecto (véase la Sección 3.3.1, en página 99). Tanto BDE como FireDAC cuentan con herramientas específicas para gestionar este tipo de archivos, en el caso de FireDAC dicha herramienta es el **FIREDAC EXPLORER**, descrita en el tercer capítulo.

Aunque podríamos usar las herramientas indicadas para, manualmente, transcribir los alias de BDE a definiciones de conexión FireDAC, esta es una tarea que puede completarse de forma automática. En el menú **CONNECTION** del **FIREDAC EXPLORER** encontraremos la opción **IMPORT BDE ALIASES**. Esta da paso a un cuadro de diálogo como el que aparece en primer plano en la Figura D.6. No tenemos más que seleccionar los alias que nos interesen y la conversión se completará en un instante.

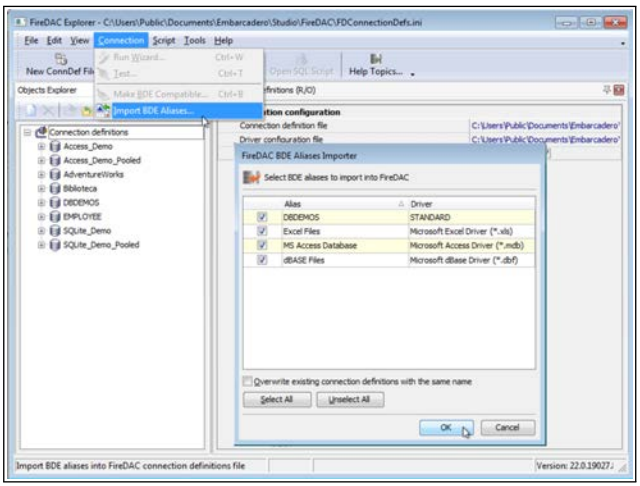


Figura D.6 HERRAMIENTA PARA IMPORTAR ALIAS

D.3 Herramientas de conversión

La transición de BDE a FireDAC en un proyecto de cierto tamaño, con decenas de módulos de código y formularios, puede resultar una tarea realmente tediosa. La mayoría de los pasos son relativamente sencillos, pero mecánicos y repetitivos. Esto hace que sea fácil su automatización, no necesitamos más que una herramienta apropiada de búsqueda y sustitución.

RAD Studio incorpora una utilidad llamada `refind` cuyo objetivo es evaluar una expresión de búsqueda y, en aquellos casos en que haya coincidencias, sustituir el contenido original por el obtenido de otra expresión regular. Podemos utilizarla desde la línea de comandos de acorde a la siguiente sintaxis general:

```
refind archivos /P:búsqueda /R:sustitución
```

Como `archivos` puede utilizarse una lista de nombres de archivo, así como incluir caracteres comodín. La opción `/P` establece la expresión que se ha de localizar en los archivos, mientras que `/R` indica la expresión de sustitución. Por ejemplo:

```
refind *.dfm /P:TTable /R:TFDTable
```

En la práctica, para convertir un proyecto completo nos será más cómodo crear un archivo de reglas para las acciones a ejecutar, entregándolo como parámetro a `refind` mediante la opción[1] `/X`. Por ejemplo:

```
refind *.dfm /X:ArchivoReglas.txt
```

Las reglas que pueden definirse son de tres tipos, identificados por los siguientes marcadores al inicio de cada línea del archivo de reglas:

- `#unuse`: Permite eliminar referencias a módulos externos de la cláusula `uses` de los módulos procesados. La regla `#unuse BDE`, por ejemplo, eliminaría las referencias al módulo `BDE`. Esta es una operación necesaria ya dicho módulo no existe en FireDAC.
- `#remove`: Sirve para eliminar las sentencias en que se utiliza una determinada propiedad, cuyo nombre se entrega como argumento. Por ejemplo, con la regla `#remove WorkingDir` eliminaríamos las referencias a dicha propiedad de BDE.
- `#migrate`: Esta regla precisa dos argumentos: el nombre de un elemento de BDE y el nombre del elemento correspondiente en FireDAC. Opcionalmente se puede facilitar, tras una coma, una lista de módulos a añadir a la cláusula `uses`. Por ejemplo, la regla `#migrate TUpdateSQL -> TFDUpdateSQL, FireDAC.Comp.Client` cambiaría las referencias a `TUpdateSQL` por `TFDUpdateSQL`, agregando la referencia necesaria al módulo `FireDAC.Comp.Client`.

En el subdirectorio `BDE2FDMigration` del directorio de la herramienta `refind` se facilita el archivo `FireDAC_Migrate_BDE`. Este contiene todas las reglas necesarias para convertir un proyecto basado en BDE para utilizar FireDAC en su lugar. Podemos usar este archivo de reglas con cualquier proyecto propio, automatizando una gran parte del trabajo. Tras este paso habrá que crear la configuración de conexión a la base de datos, agregar al proyecto otros componentes que sean precisos, por ejemplo el `TFDPhysXXXLink` apropiado, y

[1] Puedes encontrar la lista completa de opciones aceptadas por esta herramienta en el archivo `readme.txt` que la acompaña. Por defecto se encuentra en el directorio `Samples/Object Pascal/Database/FireDAC/Tool/reFind` de la carpeta donde se hayan instalado los ejemplos.

realizar los ajustes manuales que sean necesarios, según lo descrito en la sección previa. También podemos ajustar las reglas del archivo mencionado para llevar a cabo acciones adicionales, según nuestras necesidades.

D.4 Resumen

En este último apéndice del libro se han facilitado algunas directrices generales cuyo objetivo es facilitarnos el proceso de conversión de proyectos Delphi en los que usa BDE, una tecnología de acceso a datos totalmente obsoleta, para comenzar a utilizar FireDAC.

Gracias a herramientas como el **FIREDAC EXPLORER**, que permite importar alias BDE a FireDAC, y la utilidad `refind` junto con el archivo de reglas facilitado con Delphi, una buena parte del trabajo de conversión puede completarse de forma automática. No obstante, en proyectos reales generalmente siempre se necesitarán ajustes manuales.

Una alternativa, que normalmente debería ser nuestra última opción, sería la instalación en nuestra actual versión de Delphi de BDE y los componentes de BDE. El proceso se describió al inicio del apéndice.

Índice alfabético

.dfm, 130
.dpr, 545
.fmx, 63
.pas, 63, 545
$R+, 565

abstract, 601
Access, 79
Actions, 433
Active, 71
Add all fields, 106
Add Breakpoint, 535
Add Featured Files, 173
Add field, 106
Add New Connection, 69
Add Platform, 531
Add Watch, 539
AddConnectionDef, 270
AddElement, 386
AddPair, 386
ADO, 48
ADOExpress, 49
AES, 77
afAll, 335
afDisable, 335
afTruncate, 335
Aggregates, 105
Aislamiento, 313
Align, 72
Analyze, 200
and, 545, 577
ANSI, 243

AnsiChar, 556
AnsiString, 238, 557
AnyDAC, 53
Append, 282
AppendRecord, 218
ApplyUpdates, 340
array, 567
array of array, 570
AsDateTime, 115
AsDouble, 386
AsInt, 386
AsInt64, 386
Atomicidad, 312
atributos, 594
Auth, 486
Authentication, 486
AutoCommit, 316
AutoConnect, 336
AutoFetchAll, 335
AutoStart, 316, 397, 413
AutoStop, 316

backtoolbutton, 167
Backup, 226
BaseURL, 441
BDE, 44, 498
begin, 544, 581
Bind Visually, 74
Blank Application, 62
Boolean, 557
Borland Database Engine, 498
break, 584
Build Configurations, 533
Byte, 555
ByteBool, 558

CachedUpdates, 283, 340
Call Stack, 540
CancelUpdates, 341
Cardinal, 555
case, 580, 581
casting, 579
CatalogName, 302
ChangeAlerter, 326
ChangeAlertName, 326
ChangeCount, 342
ChannelQueueSize, 397
ChannelResponseTimeout, 397
Char, 557
CharacterSet, 367
CheckRequired, 282
CheckUpdatable, 282
class, 594
class var, 613
Clear Data, 214
Client, 72, 441
CliObj, 227
CloseConnectionDef, 270
CLX, 50
code point, 232
COM, 48, 182, 236
Combine, 172
Commit, 317
Commit from Project, 647
Commit window, 647
CommitUpdates, 343
conjunto, 574
Connected, 67
Connection, 71
Connection Editor, 66
ConnectionCount, 270
ConnectionDefName, 95
Connections, 270
Consistencia, 313
Constraints, 105
constructor, 595
ContentType, 441
continue, 584
Control, 140
Controls, 551
Convert, 244
CopyDataSet, 219
Count (TJSONArray), 386
CountUpdateRecords, 288
Create, 595
CreateDataSet, 217
CSV, 80, 195
Currency, 556
CustomContrainst, 107

Data Explorer, 69, 91
    Add New Connection, 92

Database, 226
DatabaseObj, 226
DataDef, 255
DataField, 134
DataSets, 198
DataSnap, 52
DataSnap REST Application, 426
DataSource, 140
Date, 115
DateUtils, 115
DaysBetween, 115
dbExpress, 50
dbGo, 48
DCOM, 53
DDL, 210
dec, 608
Default, 243
Delete, 282
DeleteConnectionDef, 270
DeleteSQL, 292
Delimiter, 201
Delta, 340
DES, 77
DestDatabase, 226
DestDatabaseObj, 226
Destroy, 595
destructor, 595
DetailCascade, 356
DetailFields, 147
DHCP, 265
DIF, 80
Differences, 648
Direction, 140
DirectX, 126
DisconnectAction, 318
Dispose, 608
div, 577
do, 582
Double, 555
downto, 583
DriverIntf, 663
DriverLink, 226, 370
DriverName, 66
DropDatabase, 367
DSContext, 434
DSHostName, 434
Durabilidad, 313

EAccessViolation, 590
Edit, 282
else, 581
EMS, 374, 470
EnableDelete, 282
EnableInsert, 282
EnableNested, 319
EnableUpdate, 282
Encoding, 255
end, 581
entry, 544, 558
enumeraciones con ámbito, 563
EstimatedByteSize, 387
Evaluate/Modify, 539
excepción, 589
except, 589
Exception, 590
ExecSQL, 192, 218, 302
Execute, 104, 196, 441, 664
ExecuteAll, 306
ExecuteScript, 303
ExecuteStep, 306
Explorador de datos, 91
Extended, 555
Extract Method, 631

FDCode, 349
FDConnectionDefs.ini, 96
FDManager, 270, 664
FDObjName, 349
FetchAll, 276
FetchNext, 276
FetchOptions, 276
    Mode, 276
FetchRowSQL, 293
FieldByName, 349
FieldDefs, 217
FieldName, 140
Fields, 105
Filter, 336
FilterChanges, 343
Filtered, 337
finalization, 548
finally, 591

FireDAC, 53
Explorer, 97
Monitor, 101
FireDACJSONReflect, 459
Firemonkey, 63
FireUI, 63
FixedInt, 555
FixedUInt, 555
FMX, 63
FMX.Controls, 552
FMX.Graphics, 600
FMX.Platform, 551
FMX.Types, 552
for, 582
Format Project Sources, 629
Format Source, 629
FormatSettings, 201
Formatter, 629
FreeMem, 608

gbak, 368
GDI, 126
Gestor de proyectos, 506
GetBufferEncoding, 245
GetByteCount, 244
GetBytes, 244
GetConnectionDefNames, 270
GetConnectionDefParams, 270
GetDocumentsPath, 172
GetEncoding, 243
GetEnumerator, 386
GetListCount, 459
GetListValue, 459
GetListValueByName, 459
GetMem, 608
GetPreamble, 246
GetString, 244
GetValue, 386
gfix, 368
Git, 634
git
.gitattributes, 653
.gitignore, 653
Carpeta .git, 640
clone, 640
commit, 650
config, 637
git.exe, 639
init, 645
pull, 651
push, 652
repositorio, 640
goto, 591
GPU, 126
gsec, 368
GUI, 43

HDR, 183
High, 570, 587
History, 631
HTMLDoc, 431
HTMLFile, 431

ibconsole, 159
IBLite, 77
IBX, 47
IDAPI, 44
IDE Insight, 64, 505
identificadores, 558
if, 580
if-then-else, 580
implementation, 549
Import BDE Aliases, 99, 665
in, 575, 578
inc, 608
Indexes, 105
inherited, 598, 611
INI, 100
initialization, 548
InsertSQL, 292
Inspect, 538
Install Packages, 660
InstanceName, 364
Int64, 555
Integer, 555
InterBase, 47
InterBase Lite, 77
InterBase Manager, 158
InterBase ToGo, 41, 77
interface, 549
intersección, 578
InTransaction, 317

IPPeerClient, 417
Isolation, 318
ItemAppearance, 72
ItemHeader.Text, 76
Items (TJSONArray), 386

JSON, 53
JSONValue, 441

KeyFields, 291
Kind, 324
Kylix, 50

label, 592
Length, 238, 570, 587
LifeCycle, 397
linkBidirectional, 140
linkControlToData, 140
linkDataToControl, 140
ListAdd, 459
ListItemRightDetail, 72
Live Bindings Designer, 74
LiveBindings Wizard, 110
lmNone, 320
lmOptimistic, 320
lmPessimistic, 320
LoadFromFile, 215, 217, 246
Local InterBase Server, 157
LocalSQL, 216
LockMode, 320
LockPoint, 321
LockSQL, 293
LockWait, 321
LoginPrompt, 67, 164
LongInt, 555
LongWord, 555
Low, 570, 587
lpDeferred, 321
lpImmediate, 321

módulo de datos, 64, 90
módulo de programa, 545
módulo estándar, 545
Macros, 105
MasterFields, 147
MasterSource, 147
matriz dinámica, 569
MaxThreads, 398
Message, 349
Method, 441
METHODINFO, 395
MethodType, 434
MIDAS, 52
mod, 577
Mode, 183, 276
ModifySQL, 292
moldeado de tipo, 579
MonitorBy, 103
Multi Device Preview, 515
multi-capa, 42
Multi-Device Application, 62

Names, 324
NativeInt, 555
NativeUInt, 555
New, 608
New field, 107
Next record set, 104
nil, 607
not, 545, 577
notación exponencial, 559

Object Inspector, 65
ODBC, 44
    ODBCAdvanced, 188
    ODBCDriver, 188
Offline, 335
Offlined, 335
OLE, 236
OLE DB, 48
OnAlert, 325
OnCalcFields, 108
OnConsoleGet, 306
OnConsolePut, 306
OnGetClass, 397
OnGetDataSet, 198
OnHTMLTag, 431
OnOutput, 101
OnProgress, 227
OnReconcileError, 343, 349
OnTimeout, 325
OnUpdateError, 350

OnUserAuthenticate, 380
OnUserAuthorize, 380
OpenGL, 126
OpenMode, 367
Options, 324
or, 545, 577
OSAuthent, 266, 366

PageCount, 227
Pairs, 386
parámetros, 586
parámetros (consulta), 109
ParamByName, 109
Params, 105, 109, 441
    TxOptions, 318
Params.Database, 66
Params.OpenMode, 67
Parse, 386
ParseJSONValue, 386
Password, 266, 366
PathInfo, 435
PersistentClass, 397
PoolCleanupTimeout, 269
Pooled, 268
PoolExpireTimeout, 269
PoolMaximumItems, 269
PoolSize, 398
Port, 364, 397
Post, 282
private, 595
program, 546
Program Pause, 534
Program Reset, 537
protected, 595, 596
Protocol, 364
Provider, 87
public, 595, 596
published, 595, 596
punteros, 558
PWideChar, 237

Query Editor, 104

RAD, 89
raise, 590
Range Checking, 565
RDBMS, 40
read, 603
Reader, 196
ReadOnly, 183
    TxOptions, 318
Reconcile, 342
RecordFormat, 255
RecsMax, 277
RecsSkip, 277
Refactor, 630
refind, 666
Register, 325
RegisterResource, 492
registros, 571
Remaining, 227
Remove, 386
RemovePair, 386
repeat, 582
Resource, 441
ResourceName, 491
ResourceOptions, 336
ResourceSuffix, 492
respositorio, 634
REST, 42, 53, 374, 380
RESTContext, 434
RESTDebugger, 440
Result, 586
Revert, 648
RevertRecord, 342
Rollback, 317
Round, 579
RowsetSize, 276
RTTI, 395, 549
Run, 534
Run to Cursor, 537
Run Until Return, 537
Run Without Debugging, 534

Save current desktop, 504
Save To File, 214
SavePoint, 342
SaveToFile, 226, 246
SchemaAdapter, 356
SchemaName, 302
SCOPEDENUMS, 563
ScriptDialog, 306

ScriptOptions, 303
sealed, 617
Search, 629
self, 612
Separator, 201
Server, 364
Session, 664
Set debug desktop, 504
set of, 574
SetLength, 569
shl, 577
ShortInt, 555
shr, 577
Single, 555
SmallInt, 555
SOAP, 380
sobrecarga de métodos, 585
SOHO, 79
SQL
    GROUP BY, 188
    SUM, 188
SQL Links, 44
SQL Script, 67
SQLite, 78
SQLite database, 69
SQLScriptFileName, 303
Start, 397
StartTransaction, 317
Step Over, 537
Stop, 397
StopOptions, 316
StoredProcName, 302
String, 557
StringElementSize, 238
StyleLookup, 167
System.Character, 557
System.JSON, 386
System.Types, 552, 619

TableName, 71, 221
TActionList, 169
TAlignLayout, 560
Target Platforms, 530
TAutoIncField, 664
TBackendAuth, 486
TBackendGroups, 485
TBackEndPoint, 485
TBackendQuery, 485
TBackendUsers, 485
TBaseLinkingBindSource, 136
TBindDataSourceDB, 137
TBindDataSourceDBX, 137
TBindingsList, 136, 137
TBindNavigator, 136, 145
TBindScope, 136
TBindScopeDB, 136
TBindSourceDB, 74, 136
TBitmap, 600
TBlobStrem, 664
TCharacter, 557
TCheckBox, 115
TClientDataSet, 415
TComponent, 90, 395
TCustomBindSourceDB, 137
TCustomDSRESTServerTransport, 396
TCustomRESTResponse, 486
TCustomSQLDataSet, 85
TDatabase, 44, 664
TDataModule, 90, 395
TDataSet, 85
TDataSetProvider, 415
TDataSource, 131
TDateEdit, 115
TDBEdit, 133
TDBMemo, 133
TDBNavigator, 133
TDSAdminRestClient, 464
TDSAuthenticationManager, 380
TDSHTTPServerTransport, 396
TDSHTTPWebDispatcher, 430
TDSProviderConnection, 415
TDSProxyGenerator, 431, 448
TDSRestConnection, 432
TDSRestMetaDataProvider, 432
TDSServerClass, 395
TDSServerMetaDataProvider, 431
TDSServerModule, 395, 416
TDSServerTransport, 396
TDSTCPServerTransport, 396
Templates, 627
TEMSClientAPI, 486
TEMSProvider, 484

TEncoding, 243
TEndpointRequest, 493
TEndpointResponse, 493
TextFileEncoding, 253
TFDAdaptedDataSet, 86
TFDAutoIncField, 664
TFDBatchMove, 195
TFDBatchMoveDataSetReader, 196
TFDBatchMoveDataSetWriter, 196
TFDBatchMoveSQLReader, 196
TFDBatchMoveSQLWriter, 196
TFDBatchMoveTextReader, 196, 201
TFDBatchMoveTextWriter, 196
TFDBlobStream, 664
TFDCommand, 86
TFDConnection, 64
    Params, 183
TFDCustomConnection, 270
TFDCustomManager, 270
TFDDataMove, 253
TFDDataSet, 85, 137, 215
TFDDatSRowState, 349
TFDEventAlerter, 323
TFDGUIxScriptDialog, 306
TFDGUIxWaitCursor, 71, 86
TFDIBBackup, 161, 371
TFDIBConfig, 87, 161, 371
TFDIBDump, 371
TFDIBRestore, 161, 371
TFDIBSecurity, 161, 371
TFDIBValidate, 161, 371
TFDJSONDataSets, 459
TFDJSONDataSetsReader, 459
TFDJSONDataSetsWriter, 459
TFDJSONDeltas, 459
TFDJSONDeltasApplyUpdates, 459
TFDJSONDeltasReader, 459
TFDJSONDeltasWriter, 459
TFDLocalSQL, 80, 198
TFDMemTable, 80, 85
TFDMoniCustomClientLink, 101
TFDMoniFlatFileClientLink, 102
TFDMoniRemoteClientLink, 102
TFDMSAccessService, 87
TFDParam, 664
TFDPhysDB2DriverLink, 83
TFDPhysDriver, 663
TFDPhysDriverLink, 663
TFDPhysIBDriverLink, 362
TFDPhysODBCDriverLink, 84, 189, 662
TFDPhysOracleDriverLink, 83
TFDPhysSQLiteDriverLink, 71, 83
TFDPhysTDBXDriverLink, 84, 662
TFDQuery, 85
TFDSchemaAdapter, 355
TFDScript, 303
TFDScriptOptions, 304
TFDSQLiteBackup, 87, 226
TFDSQLScript, 303
TFDStanStorageXLink, 459
TFDStoredProc, 86, 95, 302, 406
TFDTable, 71, 85
TFDTransaction, 86
TFDUpdateSQL, 292
TField, 106
TGrid, 111, 116
then, 580
TIBCustomDataSet, 85
Timeout, 325
TJSONArray, 386
TJSONFalse, 386
TJSONNull, 387
TJSONNumber, 386
TJSONObject, 386
TJSONPair, 387
TJSONString, 386
    Value, 386
TJSONTrue, 386
TJSONValue, 386
TJumpList, 128
TLabel, 116
TLabeledEdit, 144
TLinkControlToField, 137
TLinkGridToDataSource, 137
TLinkListControlToField, 137
TListView, 72
    Item.Text, 168
    ItemHeader.Text, 168
to, 583
To-Do List, 553
ToBytes, 387
TODO, 553

Toggle Breakpoint, 534
Tool Palette, 64
TOpenDialog, 247
TPageProducer, 431
TPanel, 115
TParam, 664
TPath, 172
TPrototypeBindSource, 142
TQuery, 45, 664
Trace Into, 536
Trace Options, 113
Tracing, 113
Transacción
    ACID, 313
    Aislamiento, 313
    Atomicidad, 312
    Consistencia, 313
    Durabilidad, 313
TRESTClient, 440
TRESTRequest, 440
TRESTResponse, 440
Trunc, 579
try, 589
TryGetValue, 386
TSaveDialog, 247
TSession, 664
TSQLConnection, 410
TStoredProc, 664
TStringDynArray, 619
TStrings, 523
TTabControl, 165
    ActiveTab, 169
    Add TTabItem, 166
TTabItem, 166
TTable, 45, 664
TTabPosition, 560
TTaskbar, 128
TTransportFilter, 380
TUpdateSQL, 664
TWebActionItem, 433
TWebFileDispatcher, 430
TWebModule, 428
TxOptions, 315
type, 560
TypeInfo, 492
Types, 551
UInt64, 555
UndoLastChange, 342
unión, 578
Unicode, 230, 243
UnicodeString, 237, 557
unit, 545
UnlockSQL, 293
Unregister, 325
until, 582
UpdateKind, 349
UpdateMode, 284
UpdateObject, 292
UpdateOptions, 282
    UpdateMode, 284
UpdatesPending, 342
UpdateTableName, 289
upWhereAll, 284
upWhereChanged, 284
upWhereKeyOnly, 284
URLHost, 485
URLPort, 485
Use Unit, 73, 547
UserName, 266, 366
uses, 547
Uses Permissions, 175
UTF, 234
UTF8, 243

ValidateAll, 306
ValidateStep, 306
Value, 115
Value (TJSONString), 386
Values, 386
var, 547, 553
Vcl.Controls, 552
VendorHome, 363
VendorLib, 362, 363
Version Control, 639
View, 69
View Source, 546
Views, 512

wearable, 40
WebDirectories, 431
WebDispatch, 434
WebFileExtensions, 431

while, 582
WideChar, 237, 556
WideString, 236, 557
with, 573, 592
Word, 555
WordBool, 558
WorkingDir, 665
write, 603
WriteBOM, 252
Writer, 196, 432

xiDirtyRead, 319
xiReadCommitted, 318
xiRepeatableRead, 318
xiUnspecified, 319
XML, 80
xoIfAutoStarted, 316
xoIfCmdsInactive, 316
xor, 577

Programación de aplicaciones Delphi con acceso a base de datos

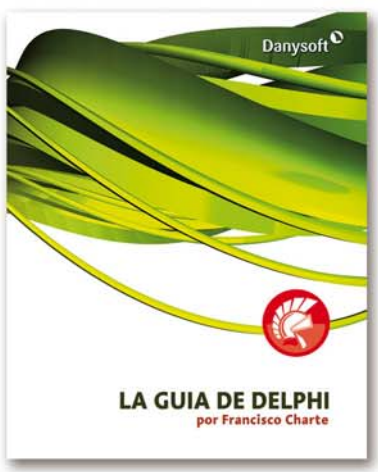
Delphi siempre se ha caracterizado, desde su primera versión, por ofrecer al desarrollador **mecanismos que simplifican el trabajo con distintas bases de datos**. Actualmente, 20 años después del lanzamiento, el acceso a bases de datos sigue siendo uno de los puntos fuertes de Delphi, así como la razón para que muchos profesionales lo elijan como entorno de trabajo.

El objetivo del libro es describir los componentes, las herramientas y los procedimientos a seguir para operar con bases de datos al crear aplicaciones Delphi en distintos escenarios. Nuestra pretensión es ofrecer al lector la información que necesitaría en cada caso concreto, explicando detalladamente cuál sería la configuración a usar y acompañando dichas explicaciones con ejercicios demostrativos que las pongan en práctica.

Como la práctica totalidad de libros sobre programación, **está pensado para ser leído prácticamente delante del ordenador**, a fin de que se pueda ir experimentando cada paso personalmente. Para facilitar este seguimiento **los proyectos propuestos como ejercicios se facilitan al lector con la entrega**. Te recomendamos que comiences por descargar dichos proyectos para tenerlos al alcance del teclado a medida que avances.

A lo largo del libro se abordarán tres contextos de trabajo diferentes, correspondiéndose estos con las tres grandes partes en que está dividido el libro: **Acceso a datos locales, Aplicaciones cliente/servidor, y Aplicaciones distribuidas**.

Al final del libro, tras los capítulos de estas tres partes, encontramos **cuatro apéndices** cuya finalidad es ayudar a aquellos desarrolladores que no estén familiarizados con el lenguaje Delphi, con su entorno y con el uso de Git como sistema de control de versiones de código fuente.



Danysoft
Haciendo visible lo invisible

Avda de la industria 4
28108 Alcobendas | Madrid | España
info@danysoft.com | www.danysoft.com

Dos de Mayo 516
Miraflores | Lima | Perú
info@danysoft.com | www.danysoft.com.pe