

## Universal Enterprise Data Connectivity for the Multi-Device World with FireDAC

Cary Jensen, Ph.D.

Jensen Data Systems, Inc.

October 2013

---

**Americas Headquarters**  
100 California Street, 12th Floor  
San Francisco, California 94111

**EMEA Headquarters**  
York House  
18 York Road  
Maidenhead, Berkshire  
SL6 1SF, United Kingdom

**Asia-Pacific Headquarters**  
Level 2  
100 Clarence Street  
Sydney, NSW 2000  
Australia

# Universal Enterprise Data Connectivity for the Multi-Device World with FireDAC

---

## EXECUTIVE SUMMARY

FireDAC is the latest addition to RAD Studio's suite of data access frameworks, and is the preferred mechanism for accessing data in RAD Studio applications going forward. In addition to being an impressively flexible component set, it is notable for its cross-platform and device independent nature, features that are consistent with Embarcadero's emphasis on targeting multiple devices with native executables from a common source code base.

This whitepaper is divided into three sections. The first section is designed to provide you with an understanding of FireDAC, including what it is and how it compares to alternative RAD Studio data access frameworks. This section also shows you how to get started with FireDAC, and also includes a number of sample projects that demonstrate some of FireDAC's more interesting features.

The second part of this paper describes how you can start using FireDAC today in your existing applications. The primary focus in this section is moving from an existing data access framework to FireDAC. This section concludes with a discussion of a hybrid solution, where you add FireDAC's advanced capabilities to existing applications without entirely replacing your current data access components.

The third and final section of this paper demonstrates FireDAC in action. Here I will introduce a series of sample projects that demonstrate typical deployments of FireDAC. These include a traditional client/server project, a multitier DataSnap server and several DataSnap client implementations, including a mobile DataSnap client for Android.

## OVERVIEW

FireDAC is a comprehensive collection of components that implement RAD Studio's traditional TDataSet interface. In this respect, it is comparable to the Borland Database Engine (BDE), dbExpress, InterBase Express, and dbGo, RAD Studio's TDataSet components for ActiveX Data Objects (ADO). Embarcadero added FireDAC to RAD Studio in the spring of 2013 after acquiring AnyDAC from DA-Soft Technologies.

FireDAC is powerful, flexible, and well designed. It is, without a doubt, RAD Studio's new data access framework of choice.

When Delphi first shipped (long before it was joined, product-wise, with C++Builder to form RAD Studio), it had one data access framework, the BDE. While the BDE was a breakthrough product in its early years, providing a fast, independent data access layer, it was cumbersome to install, used a lot of network bandwidth, and had limited support for remote database servers. Over time, it became increasingly obsolete.

The need for a new data access mechanism for Delphi became even more apparent during the development of Kylix, a Delphi-based compiler and IDE (integrated development environment) for Linux. Porting the BDE to Linux was ruled out, and dbExpress was born. dbExpress is a high speed client/server data access framework based largely on pass-through SQL (structured query language). Furthermore, it requires client-side in-memory datasets to hold the results of the read-only, unidirectional cursors returned by the dbExpress TDataSets.

dbExpress filled developers' needs for a high-performance, client/server data access framework, but it too was limited. Those developers who needed to continue support for file server databases could not use it. (That is until RAD Studio XE2 was released. That is when the dbExpress ODBC driver was first made available.) In addition, for those client/server applications that could be migrated from BDE application, the difference in architectures required significant refactoring of legacy applications.

What was really needed was a data access mechanism that could support both local and remote database technologies. If possible, that mechanism needed to provide both the speed and power of the SQL-based dbExpress while making the migration of legacy BDE application as painless as possible. Because dbExpress didn't provide the local file server support and posed significant migration hurdles, the BDE continued to be used well beyond its practical lifecycle.

Fortunately, FireDAC fulfills both of these needs, and then some. BDE developers will feel right at home, and dbExpress developers can continue to leverage the speed they found in unidirectional cursors, if they want. In addition, FireDAC adds significant new capabilities not found in any of RAD Studio's other data access frameworks.

The following section will take a deeper look at RAD Studio's TDataSet interface. In doing so, I will discuss how FireDAC implements this interface, and why this provides a convenient migration path for both BDE and dbExpress developers.

## FIREDAC AND THE TDataSet INTERFACE

FireDAC conforms to the TDataSet interface, a collection of classes and their methods and properties that support sequential data access, data awareness, and data binding in both the visual component library (VCL) and FireMonkey. The primary classes that you work with are descendants of TDataSet, from which the name for this interface derives. In FireDAC these are the FDQuery, FDStoredProc, FDMemTable, and FDTable components. All of these objects are capable of holding an open cursor to a result set, in which case one of the records (rows) in this result set is considered the *current record* (except when the result set is an empty result set). Some of these classes can also be used to perform operations on the database that do not return a result set, such as data manipulation language (DML) statements in a query, or execution of a stored procedure.

The TDataSet interface also consists of support classes that do not descend from TDataSet. These include TFields, which are used to both manipulate the contents of individual columns of a result set as well as provide metadata about those columns, classes that permit parameter binding to parameterized queries and stored procedures, and classes that support connecting to databases.

The TDataSet interface represents the RAD Studio approach to data access, and all of the data access mechanisms that have appeared in the product over the years have supported it (some more than others). If you are new to RAD Studio, or simply new to database development in RAD Studio, you might want to skip to appendix A, found at the end of this paper, where I introduce the core concepts associated with the TDataSet interface, before you continue reading this paper.

From a strictly practical standpoint, FireDAC provides a complete implementation of the TDataSet interface, and then some. As a result, if you have used a data access mechanism that supports the TDataSet interface you will feel right at home with FireDAC. This is true if the data access mechanism you used was part of the RAD Studio product, such as the BDE or InterBase Express, or one provided by a third party, such as the Advantage Delphi Components from Sybase, an SAP company, or Direct ORACLE Access (DOA) from Allround Automations.

Let me make this even clearer. If you are familiar with using the BDE to access data, you already know how to use FireDAC. FireDAC includes an FDTable component that is nearly indistinguishable from the BDE's TTable, and the FDQuery and FDStoredProc components are equally recognizable. You can point an FDTable to an active connection for a database and retrieve an editable result set by opening the table. That result set can be freely scrolled and even edited. As each edit is posted, FireDAC updates the

underlying database. Likewise, queries that return a result set can also be edited, and in most cases, FireDAC can write those changes to the underlying database without any special intervention by you.

What I am trying to say is that FireDAC makes data access easy, just like the BDE. However, ease of use is about the only thing that FireDAC shares with the BDE. FireDAC supports more connectivity than any other single TDatabase-based data access mechanism, has a modern and rock-solid internal architecture, and supports advanced features that you will truly appreciate, if the need arises. I'll talk more about these advanced features in the next section.

The dbExpress framework did not share this same level of conformity to the original BDE. For example, dbExpress TDataSets are read-only and unidirectional, which means that they cannot be bound directly to data aware controls or LiveBindings BindSourceDBs. Likewise, some of the methods and properties inherited from TDataSet by dbExpress TDataSets raise exceptions if used, including Edit, Prior, RecNo (for navigation), AppendRecord, and the like.

Some users of dbExpress might argue that the dbExpress mechanism for pass-through SQL and its use of unidirectional cursors is an appropriate way to interact with traditional database servers, and that the additional complexity of having to move that data into and out of an in-memory table is worth the effort. Fortunately, if that approach to data best suits your needs, FireDAC can do that as well. While the FDQuery, FDStoredProc, and FDTTable are very easy to use, internally they are performing operations similar to those found in dbExpress. Specifically, FDQuery employs FDCommand, FDTTableAdapter, and FDMemTable classes to perform its work. If you want, you can use these classes directly (and also incorporate ClientDataSets) to perform custom optimized operations against your database. Doing so is more complicated, but the option is there. For most applications, however, using the FDQuery to work with data is not only easier, but more than adequate for the job.

You might notice that I didn't recommend the FDTTable as the "go to" component for working with data. In practice, the FDTTable might be your ideal component for holding a result set. By default, data accessed using an FDTTable uses a FireDAC mode referred to as *Live Data Window (LDW)*. Using LDW, the FDTTable only loads only some of the underlying table's data in memory, based on a SELECT statement that it generates which takes into account filters, sort order, FindKey and Locate calls, and the like. LDW produces a number of advantages, including near instant access to the first records, frequent refreshes, limited memory consumption, and more. It does this by loading into memory only twice as many records as defined by the property FDTTable.FetchOptions.RowsetSize, fetching additional records as the current record or

other properties (filter, index, and so forth) change. Live Data Window is a very nice feature, but introduces some potential drawbacks under certain conditions. (Specifically, it cannot be used with cached updates, may require additional configuration to avoid duplicate key errors, and the like.) As a result, I personally prefer the ease of use offered by FDQuery, which like the FDStoredProc component, does not support the LDW mode.

FireDAC actually goes beyond the TDataSet interface to embrace some useful constructs that are available only in some of Delphi's data access frameworks. For example, the FireDAC TDataSets support cached updates, which is introduced in the TBDEDataSet class (a descendant of TDataSet). Most of the other TDataSets do not support cached updates. Similarly, FireDAC TDataSets support aggregates and filtered navigation (navigation to records matching the current Filter property expression using methods such as FindNext, FindLast, and so forth). These features are introduced in the TCustomClientDataSet class, and other than FireDAC, are not supported by other TDataSets. In addition, FireDAC implements a number of useful features not found in any TDataSet implementation. It supports extended expressions in locate, filter, and sort operations, maintained local indexes, virtual drivers, and more.

Now that you've learned what FireDAC can do for you, I am certain that you want to begin using it, and the following section is designed to help you do just that. This walks you through the process of creating of a new application that employs FireDAC for its data connectivity. Here you will learn the basics of creating a connection and configuring it, executing a query through the created connection, and a few extra steps necessary to get everything compiled and running. Using these steps, you are ready to create new applications using FireDAC.

## MAKING THE CONNECTION

You connect to a database using a connection component (TFDConnection), and then wire one or more FireDAC TDataSet descendants to that connection. At this high level, these steps are no different from any other data access framework based on the TDataSet interface. At a lower-level, however, the specific steps you take are uniquely FireDAC.

FireDAC supports four general mechanisms for connecting to your database. These are:

- Configure the Params property for a TFDConnection component manually at design time



- Configure the Params property for a TFDConnection component programmatically at runtime
- Create a named connection definition using the FireDAC Explorer utility. You can then select that named connection definition using the TFDConnection component at design time
- Create a connection configuration file and load the connection information from that file at runtime

These steps are not entirely independent. For example, you might use the FireDAC Explorer to create a new connection definition file, and then use that file at runtime to connect your TFDConnection to your database. Similarly, you must configure the Params property at design time using the Connection Editor, after which, you can inspect the TFDConnection.Params TString property to see what text you can use to configure the connection at runtime.

There are additional options in FireDAC, but they are somewhat specialized and I will not cover them in detail in this paper. For example, FireDAC includes a class called FDManager. The FDManager class gives you more control over connections, but it is more complicated to use than an TFDConnection. In addition, FireDAC includes two utilities, FireDAC Explorer and FireDAC Administrator. These utilities are located in RAD Studio's BIN directory, and are standalone applications that can, in addition to other tasks, permit you to define connection definition files. For more information, please refer to the FireDAC documentation.

In this section, I will focus on the configuration of the TFDConnection component using the Connection Editor.

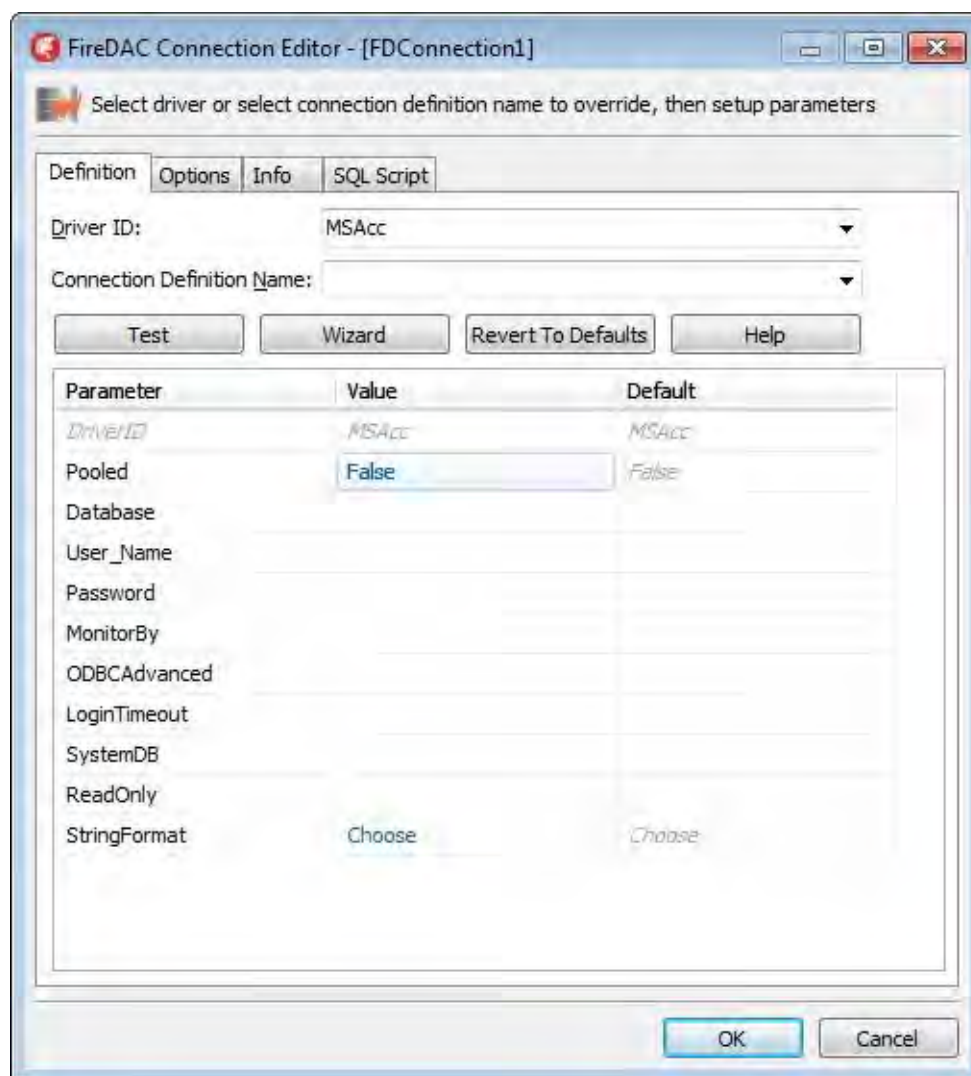
## CONFIGURING THE TFDConnection COMPONENT

Let's begin by using the TFDConnection component to configure a connection to an MS Access database. I am specifically using MS Access in this example since you can follow these steps using RAD Studio Professional or higher. The specific database that I am going to connect to is one that was installed in a samples folder by the RAD Studio installer.

Use the following steps to create, configure, and test your connection:

1. Select File | New | VCL Forms Application from Delphi's main menu to create a new project.

2. Add to this project a data module by selecting File | New | Other, and then selecting the Data Module template from the Delphi Files page of the Object Repository.
3. Add an FDConnection component from the FireDAC page of the Tool Palette to the data module.
4. Double-click the FDConnection component (or right-click it and select Connection Editor...) to view the FireDAC Connection Editor component editor. This Connection Editor, shown in the following figure, has already had its Driver ID set to MSAcc, which is why the connection parameters for the MS Access driver are shown.



If you have previously created a named connection using the FireDAC Explorer utility, you could set Connection Definition Name to that named connection. In

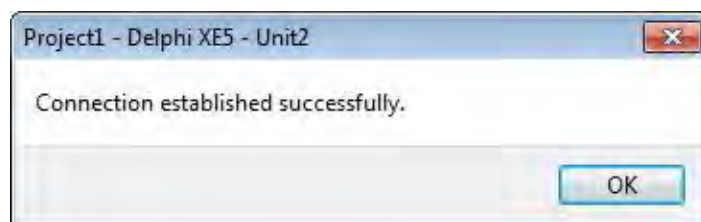


this case, however, we are going to configure a new connection. Set Driver ID to MSAcc. Once you set the value of Driver ID, the properties associated with the physical MS Access driver are displayed in the configuration pane, as seen in the preceding figure.

5. Click in the Value column of the row labeled Database, and then click on the Folder symbol to navigate to the folder where the dbdemos.mdb sample MS Access database is stored, which in the most recent versions of RAD Studio is the following directory (the version number will depend on which version of Delphi you have installed. RAD Studio XE5 is version 12):

```
C:\Users\Public\Documents\RAD Studio\12.0\Samples\Data
```

6. Since this database is not encrypted, we do not need to set the User\_Name or Password parameters. In this case, you are ready to test your connection by click the Test button on the Connection Editor.
7. FireDAC now displays a login dialog, which you use to enter your username and password before clicking OK. This dialog is always displayed when you test a connection, whether or not the underlying database is protected. In this case, leave User\_Name and Password blank and click OK. FireDAC should respond with the following dialog box, which indicates that your connection was successful.



This configuration was particularly easy since all we really needed to do was select the MS Access driver and point to the database. Depending on the database that you are connecting to, where it is located, and details about your configuration, you will likely enter more parameters than we did here.

In addition to connection parameters, the Connection Editor permits you to configure many of the FDConnection properties. These are found on the Options tab of the Connection Editor, shown in the following figure.



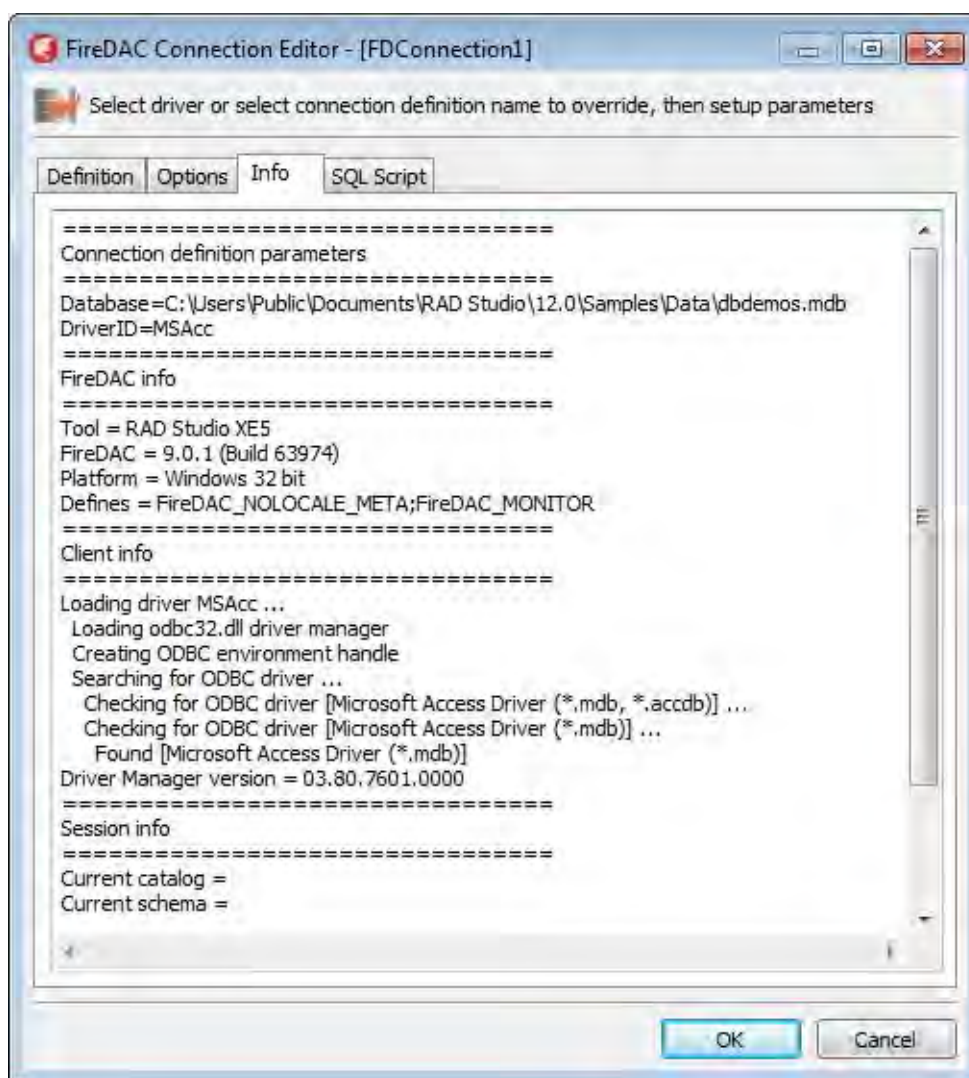
Unlike the connection parameters, which will depend on the database you are connecting to, the TFDConnection properties shown on the Options tab are the same for all connections, and control a wide variety of aspects of the connection.

Each of these options corresponds to an instance property of the FDConnection. Those properties have names such as FetchOptions, FormatOptions, ResourceOptions, and the like. Instead of configuring these properties using the Connection Editor dialog box, you can select the FDConnection component and set the corresponding options using the Object Inspector. Using the Connection Editor to configure your options is much easier than using the Object Inspector, however, since the Connection Editor provides additional assistance in assigning values to these properties.

Use FormatOptions subproperties to define how the database data types are mapped to FireDAC fields. FetchOptions control how FireDAC retrieves data from the underlying

database. The UpdateOptions property defines how FireDAC writes data back to the database, while ResourceOptions controls how FireDAC allocates its resources. Finally, use TxOptions to configure how transactions are handled.

If you have successfully tested your connection, the Info page contains details about the connection you have established. This information is especially important if you are having problems with your connection. For example, suppose that get an error when you attempt to connect to your database, and the error message indicates that you are using the wrong client driver. Display the Info tab of the FireDAC Connection Editor and examine the Client info section. There you can see which client DLL FireDAC is trying to use, as shown in the following figure.

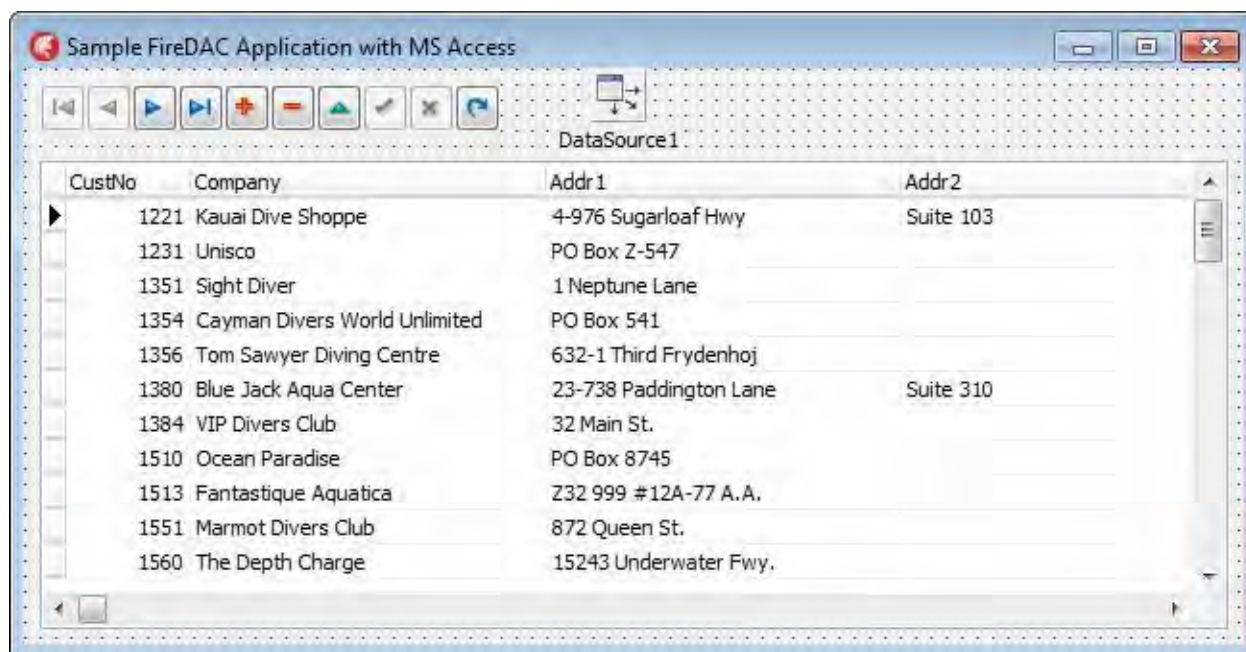


Finally, the SQL Script tab permits you to enter SQL statements that you want to execute ad hoc against this connection.

When you are done setting your connection parameters and options, close the Connection Editor by click the OK button.

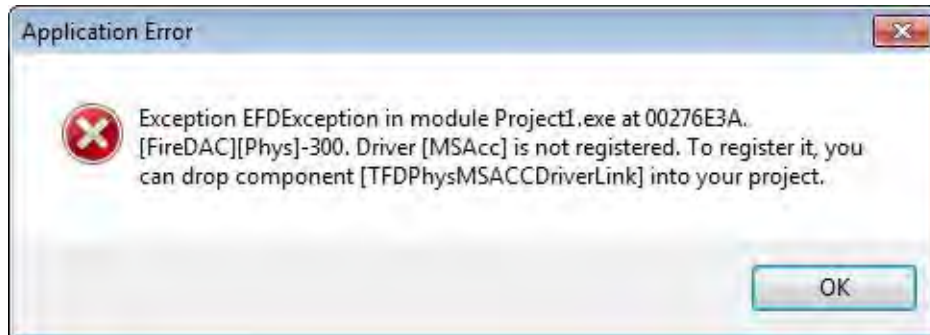
We are now ready to finish this simple example:

1. Begin by setting the LoginPrompt property of the FDConnection to False, since no username or password is necessary for this database.
2. Next, add an FDQuery to your data module. Set its SQL property to **select \* from customer**, and set its Active property to **True**. Note that you did not have to manually set the Connection property of the FDQuery component to your FDConnection component. This was done for you automatically when you dropped the FDQuery onto the data module. Nice, isn't it?
3. Return to your main form and use your data module unit by selecting File | Use Unit, selecting your data module's unit name from the displayed list.
4. Now add a DBGrid, a DBNavigator, and a DataSource component to your form. Set the DataSource property of both the DBGrid and DBNavigator to **DataSource1**, and the DataSet property of the DataSource to **DataModule2.FDQuery1** (or whatever name is appropriate for your data module and data source). Your form should now look something like that shown in the following figure.



At this point everything looks great, and if we were using one of the other data access frameworks (BDE, dbExpress, etc.) we could hit Run (F9) and the application would just

run. But there is a little quirk with FireDAC, and that is that it requires some resources that are in units that do not appear in your uses clause by default. As a result, if you hit run now, you will get a runtime error like the one shown in the following figure.



As you can see from this exception, FireDAC is telling you to place an `FDPhysMSACCDriverLink` component onto your form. There is another component that you need to add as well, and that is the `FDGUIxWaitCursor`. You will complete this simple project using the following two steps:

1. Place an `FDPhysMSACCDriverLink` component from the FireDAC Links tab of the Tool Palette onto your data module.
2. Next, place `FDGUIxWaitCursor`. You can find this component on the FireDAC UI tab of the Tool Palette.

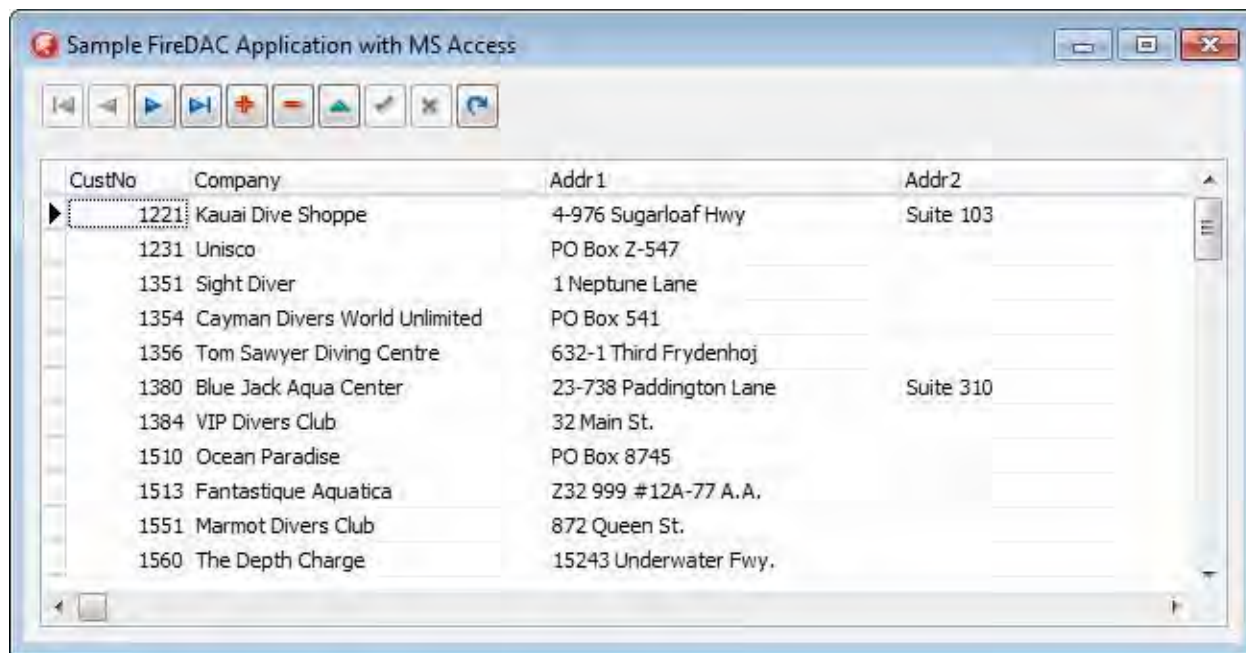
Here is the interesting part. FireDAC doesn't really need these components, and you do not need to set any of their properties either. What FireDAC *really* wants is the units associated with these components in your uses clause so that those units get initialized. The `FDPhysMSACCDriverLink` component causes the insertion of the `FireDAC.Stan.Intf`, `FireDAC.Phys`, and `FireDAC.Phys.MSAcc` units to your uses clause (at least in XE5. A distinctly different set of units are added to the uses clause in earlier versions). Similarly, the `FDGUIxWaitCursor` component results in the addition of the following units: `FireDAC.UI.Intf`, `FireDAC.VCLUI.Wait`, and `FireDAC.Comp.UI` (again, in XE5).

With respect to these last two components that we've added to the project, since it is the underlying units that FireDAC really wants, you have two options. One is to place these components in your data module (or on one of your forms: it really doesn't matter, so long as these units appear in at least one unit in your project source) and simply leave them there. The other is to place these components, and then save or compile your application to get the required units added to the associated uses clause, after which you can harmlessly delete the components.



Note: If your application employs more than one driver type, you will need to add the `FDPhysxxxxDriverLink` component for each additional driver.

Having added the necessary components, you can now press F9 and the application will run. After a moment the main form is displayed, looking something like that shown in the following figure.



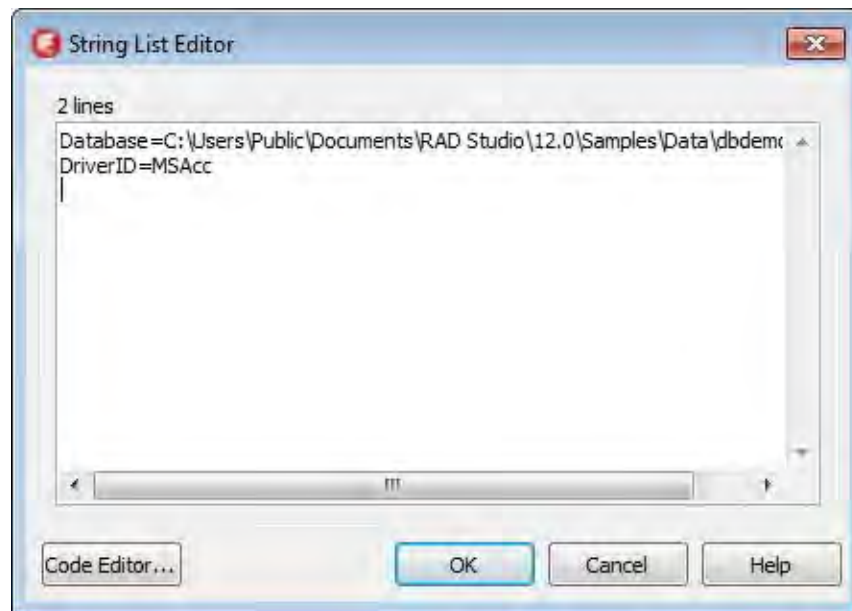
This data can now be viewed, navigated, and edited in a manner similar to that supported by the BDE's `TTable` component, even though this data is associated with a SQL query. (In most of Delphi's data access mechanisms, queries are not directly editable, at least not by default.) Since we did not enable cached updates, edits applied to the data shown on this form are written to the `dbdemos.mdb` database on a record-by-record basis.

## DEFINING A CONNECTION USING `TFDConnection.Params`

By default, the configuration you define in the Connection Editor is stored in the `Params` property of the `FDConnection`. You can easily see this in the project you just created by using the following steps:

1. With your data module displayed in the designer, select the `FDConnection` component.
2. Select the `Params` property in the Object Inspector and click the ellipsis that appears in order to open the String List Editor, shown in the following figure.





In this case, the String List Editor lists only two parameters: Database and DriverID. Remember, however, that this was a very simple configuration. We are connecting to a file server database running on the local machine. When you connect to a remote database server, there are often additional parameters that you will need to set, such as connection pooling, OS authentication, role name, and character set, to name a few examples. In other words, in most cases a successful connection requires more parameters than did this application.

Nonetheless, the bottom line is that those parameters that you set using the Definition tab of the Connection Editor are written to the Params TStrings property, and those can be easily viewed by viewing the property editor of the Params property. Similarly, those properties that you set using the Options tab of the FireDAC Connection Editor are written to the corresponding instance properties of the FDConnection.

3. Once you know the connection parameters for your database driver, you can enter these values into the Params property editor manually at design time or assign them to the Params property at runtime. For example, consider the values displayed in the String List Editor in preceding figure. The following OnCreate event handler uses these values to configure the FDConnection at runtime. And, while we are at it, I have calculated the location of the sample data using the CompilerVersion constant, which in RAD Studio XE5 is 26. When I subtract 14 from CompilerVersion, it returns the XE5 version, which is 12. (This code will not work for a future release if CompilerVersion includes decimal places.)

```
procedure TDataModule2.DataModuleCreate(Sender: TObject);  
var  
    DataDir: string;  
begin  
    DataDir := 'C:\Users\Public\Documents\RAD Studio\' +  
                FloatToStr(CompilerVersion - 14) +  
                '.0\Samples\Data\';  
    FDConnection1.Params.Clear;  
    FDConnection1.Params.Add('Database=' + DataDir + 'dbdemos.db');  
    FDConnection1.Params.Add('User_Name=sysdba');  
    FDConnection1.Params.Add('DriverID=MSAcc');  
    FDQuery1.Open;  
end;
```

Yes, it really is that easy. You can now use these basic steps to create new applications that use FireDAC as their data access mechanism, or to add database support to applications that do not currently have that capability.

## FIREDAC: POWER AND PRACTICALITY

While FireDAC does an exceptional job of supporting the TDataSet interface, going so far as to introduce some of the advanced features found in only some of the TDataSet implementations, there is much more to FireDAC than simple TDataSet conformity. FireDAC supports an exceptional collection of features and capabilities that make it the clear choice for implementing data access in RAD Studio applications.

In the following sections, I am going to highlight some of the features that make FireDAC really shine.

### CROSS-PLATFORM SUPPORT

FireDAC is supported on all of RAD Studio's platforms. You can use FireDAC in 32 and 64 bit Windows applications, OS X applications, iOS and iOS simulator applications, and Android applications.

To what extent a given platform is supported depends to some extent on the availability of an appropriate client library. For example, all databases are supported on the Windows platform, in part because every database vendor creates a client API for Windows. There are few client APIs, however, for the mobile platform. For example, both

InterBase and SQLite are supported on both iOS and Android. However, many databases do not publish a client library for the iOS or Android ARM platforms.

The good news is that even though some platforms have limited client libraries, you can always use DataSnap to serve data to your mobile devices. Using DataSnap to access data on mobile devices is covered later in this paper.

## EXCEPTIONAL SUPPORT FOR DATABASES

FireDAC provides drivers for almost every major relational database, both commercial and open source. On the commercial side, you find native support for Oracle, IBM DB2, MS SQL Server, InterBase, SAP Sybase SQL Anywhere, and Advantage Database Server from Sybase. Open source databases natively supported by FireDAC include MySQL, SQLite, PostgreSQL, and Firebird.

If you don't find your particular database in that list, there's no need to worry. FireDAC supports two bridging drivers, one for ODBC (open database connectivity) and another for dbExpress. Using the FireDAC ODBC driver permits you to work with virtually any relational database in existence, as ODBC is about as universal as it gets. Even COBOL data files are supported by ODBC.

What's particularly interesting about FireDAC's native drivers is that many of these provide you with access to features specific to the associated database. For example, FireDAC supports the return of multiple result sets from a SQL command execution from those databases that support that feature, such as MS SQL Server, Oracle, and PostgreSQL. Likewise, FireDAC's native drivers support database alerts (notifications or events).

It should be noted that full support for the databases listed above requires that you have an Enterprise-level license or higher, or that you have purchased the FireDAC Client/Server Add-On Pack for Professional-level RAD Studio products. RAD Studio Professional (as well as Delphi Professional and C++Builder Professional) are licensed only for local file server database access, such as MS Access, SQLite, and Advantage Local Server.

## FLEXIBLE QUERIES THROUGH DYNAMIC SQL

Dynamic SQL permits you to write flexible SQL that includes variables, conditional substitution, ODBC-like escape function, and macro expansion. Dynamic SQL is implemented through a SQL command preprocessor that can perform a variety of manipulations on your SQL before it sends that SQL to the underlying database.

One of more useful applications of dynamic SQL is to permit you to write one set of queries that can be executed against a variety of databases, even when those databases support different dialects of SQL. For example, the {`CONVERT(...)`} escape function will be expanded into the `TO_CHAR` keyword when executing against an Oracle database, but into `CONVERT` when connected to Microsoft SQL Server. Statements that use conditional substitution look similar to the {`$IF`} conditional compilation compiler directives in the Delphi language, though the actual substitution is not performed at compile time. Instead, this operation occurs at runtime, and is performed by the command pre-processor, right before the query is submitted to the underlying database.

Dynamic SQL has utility beyond supporting multiple databases. For example, the use of variable symbols in your SQL statements permits you to write queries whose tables, fields, or `WHERE` clause predicates are not known until runtime. All you need to do is ensure that you have bound valid values to those variables before executing the query, and the command preprocessor will take care of the updating the resulting SQL.

Several projects in the code samples associated with this whitepaper make use of dynamic SQL. The `VariableSubstitution` project, whose main form is shown in the following figure, dynamically constructs fields list and table name for a query using variable substitution.

CUSTNO	COMPANY	CITY	STATE	COUNTRY
1221	Kauai Dive	Kapaa Kau	HI	US
1231	Unisco	Freeport		Bahamas
1351	Sight Diver	Kato Paph		Cyprus
1354	Cayman Di	Grand Cay		British Wes
1356	Tom Sawy	Christianst	St. Croix	US Virgin I
1380	Blue Jack A	Waipahu	HI	US
1384	VIP Divers	Christianst	St. Croix	US Virgin I
1510	Ocean Par	Kailua-Kon	HI	US
1513	Fantastiqu	Bogota		Columbia
1551	Marmot D	Kitchener	Ontario	Canada
1560	The Depth	Marathon	FL	US
1563	Blue Sport	Giribaldi	OR	US
1624	Makai SCU	Kailua-Kon	HI	US
1645	Action Clul	Sarasota	FL	US
1651	Jamaica SC	Negril	Jamaica	West Indie
1680	Island Find	St Simons	GA	US
1984	Adventure	Belize City		Belize
2118	Blue Sport	Largo	FL	US

The actual query that is executed to create the tabular view on the right side of this form is shown here:

```
SELECT &FieldList FROM &TableName
```

The code that builds the data for the field list, and assigns the values to the variables is shown here:

```
procedure TForm1.LoadData;
var
  i: Integer;
  FieldList: string;
  ListBoxItem: TListBoxItem;

procedure BuildList(Value: string);
begin
  if FieldList = '' then
    FieldList := Value
  else
    FieldList := FieldList + ', ' + Value;
end;
```

```

begin
  DataModule2.SelectQry.Close;
  for i := 0 to ListBox2.Items.Count - 1 do
    if ListBox2.ListItems[i].IsSelected then
      BuildList(ListBox2.ListItems[i].Text);
  DataModule2.SelectQry.Macros[0].AsRaw := FieldList;
  DataModule2.SelectQry.Macros[1].AsRaw := ListBox1.Selected.Text;
  DataModule2.SelectQry.Open;
end;

```

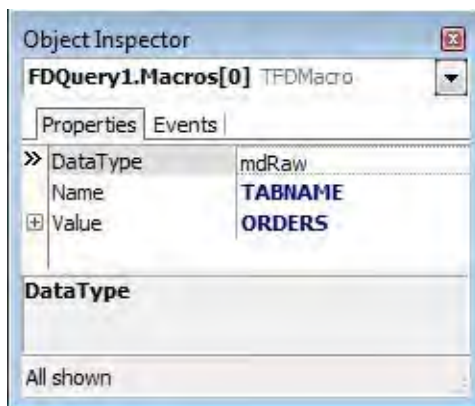
Yet another project, Macros, includes a query that employs both variable substitution and macro expansion. The following is the text of the query executed by this project. `Current_Date()` and `Extract()`, which are enclosed in a pair of matching curly braces, are FireDAC date/time functions. `&TabName`, as in the preceding example, is a variable. Finally, `{e 1988}` is a constant substitution escape sequence, which formats the value consistent with the format required by the underlying database:

```

SELECT {Current_Date()} AS Today,
       c.*
FROM &TabName c
WHERE {Extract(Year, c.SaleDate)} = {e 1988}

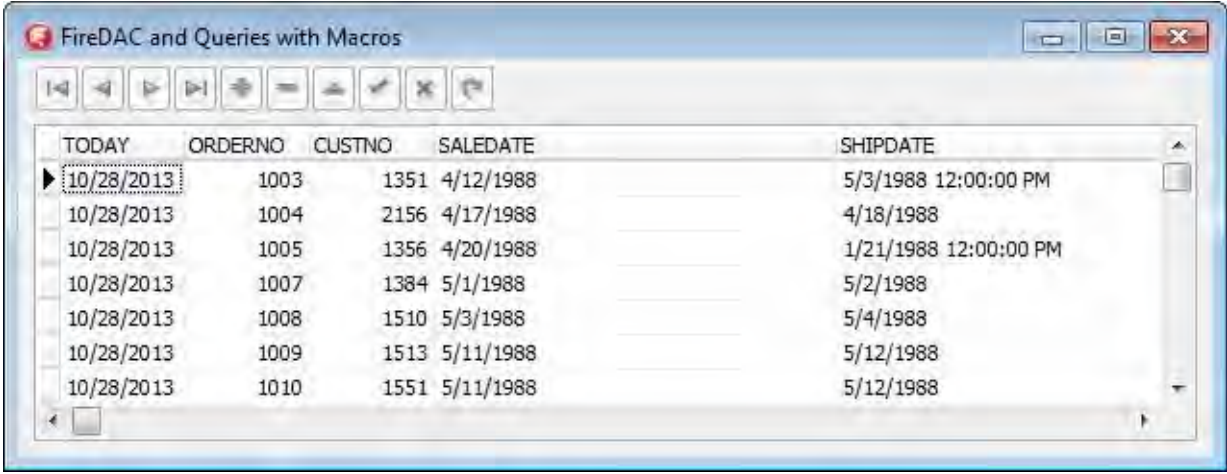
```

The following figure shows the Macros property of the `FDQuery` that executes this query, depicting the assignment of the value to the `&TabName` variable.



The result set produced by this query is shown in the following figure.





The screenshot shows a window titled "FireDAC and Queries with Macros". Inside, there is a table with the following columns: TODAY, ORDERNO, CUSTNO, SALEDATE, and SHIPDATE. The table contains seven rows of data. The first row is highlighted with a mouse cursor.

TODAY	ORDERNO	CUSTNO	SALEDATE	SHIPDATE
10/28/2013	1003	1351	4/12/1988	5/3/1988 12:00:00 PM
10/28/2013	1004	2156	4/17/1988	4/18/1988
10/28/2013	1005	1356	4/20/1988	1/21/1988 12:00:00 PM
10/28/2013	1007	1384	5/1/1988	5/2/1988
10/28/2013	1008	1510	5/3/1988	5/4/1988
10/28/2013	1009	1513	5/11/1988	5/12/1988
10/28/2013	1010	1551	5/11/1988	5/12/1988

## CACHED UPDATES

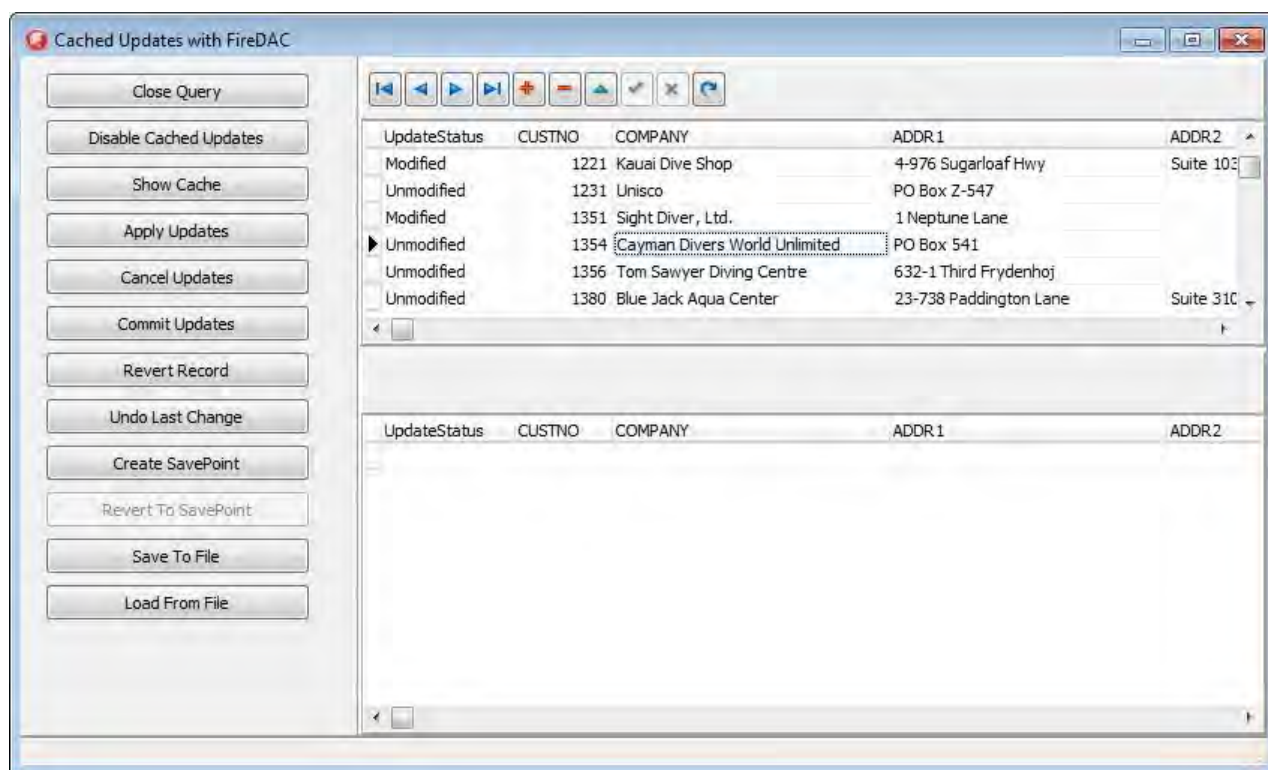
All TDataSets in FireDAC support cached updates. When enabled, changes made to records are cached in memory. That cache can then be used to review the changes, selectively revert specific changes, cancel all changes, or apply those changes still in cache to the underlying database.

FireDAC provides you modes for cached updates: the centralized and the decentralized. In the decentralized mode updates are cached individually for each dataset. When you are ready to apply the changes in the cache, you do so by calling the dataset's ApplyUpdates method.

The centralized mode provides a single change cache for two or more datasets. In order to use the centralized mode, you use an FDSchemaAdapter component, which you then assign to the SchemaAdapter property of each FireDAC TDataSet that will participate in the shared change cache. As changes are made to individual TDataSets, those changes are logged in a single cache that is implemented by the FDSchemaAdapter. When you are ready to apply the changes in the cache, you do so by calling the FDSchemaAdapters's ApplyChanges method.

The centralized mode is especially useful when you are caching updates to a master-detail relationship. In particular, this mode permits updates and deletions to be cascaded from the master dataset to the detail dataset(s). In addition, you can wrap your call to ApplyUpdates in a transaction, which you commit if the application of the changes is successful. This ensures that the changes to the master and detail tables are applied in an all-or-none fashion.

Cached updates provide a wealth of powerful options, including the ability to exercise complete control over the process of writing data to the underlying database. I spoke at length about cached updates at Embarcadero's CodeRage 8. During that talk, I demonstrated cached updates as well as other features of FireDAC, using the project whose main form is shown in the following figure. This project is named BasicCachedUpdates. There is any additional project in the code download that demonstrates the use of the FDUpdateSQL component, which can be used to customize the process of writing data to the underlying database when cached updates are being applied.



For more information on cached updates, I encourage you to search the Embarcadero Developers Network site ([edn.embarcadero.com](http://edn.embarcadero.com)) to find the CodeRage 8 replay page, from which you can view my video.

## RESULT SET PERSISTENCE

The data in FireDAC TDataSets can be written to disk or to a stream to persist that data from one session to the next without reading from a database. This feature, which is nearly identical to the SaveToFile and SaveToStream methods of the ClientDataSet, gives you added flexibility when designing your applications.

In addition, if you are employing cached updates in your FireDAC TDataSet, you will find that the change cache is also part of the option to persist a result set to a file or stream. This permits you to maintain information about changes to your data over multiple sessions and an extended period of time.

Persisting a FireDAC TDataSet is demonstrated in the BasicCachedUpdates project, whose main form is shown in the preceding figure. The following two event handlers are associated with the OnClick events of the buttons labeled Save To File and Load From File in the main form:

```
procedure TForm1.SaveToFileBtnClick(Sender: TObject);  
begin  
    FDQuery1.SaveToFile(ExtractFilePath(ParamStr(0)) +  
        'file.xml', TFDStorageFormat.sfXML);  
end;  
  
procedure TForm1.LoadFromFileBtnClick(Sender: TObject);  
begin  
    if IOUtils.TFile.Exists(ExtractFilePath(ParamStr(0)) + 'file.xml') then  
    begin  
        //Disconnect the query from the DataSource  
        DataSource1.DataSet := nil;  
        FDQuery1.LoadFromFile(ExtractFilePath(ParamStr(0)) +  
            'file.xml', TFDStorageFormat.sfXML);  
        //Reconnect the DataSource.  
        DataSource1.DataSet := FDQuery1;  
    end;  
end;
```

What's interesting about LoadFromFile is that it allows you to work with data offline. For example, after loading data from a database into an FDQuery, you can save the query results to a local file, after which the data can be loaded again without the presence of the database. Here again, FireDAC goes beyond the traditional usage of dataset persistence. Combining connection offline mode, dataset persistence, and cached updates mode your application can work disconnected from a database and persist the data between application run sessions. Later, when the database is once again available, any changes to the data can be applied to the underlying tables.

## BLAZING PERFORMANCE WITH ARRAY DML

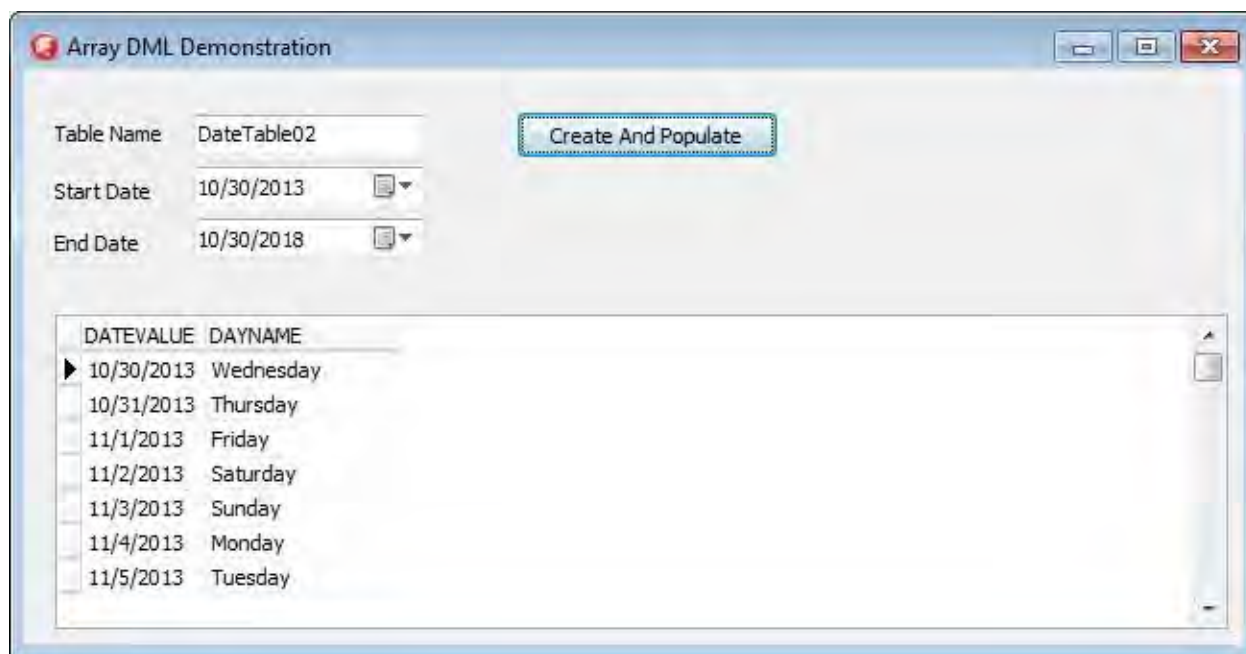
Array DML (data manipulation language, a subset of SQL), provides a mechanism that supports high-speed data manipulation using parameterized query and stored procedure execution.

To use Array DML, you create a parameterized SQL statement, along with an array of parameters. When executed, FireDAC sends both the parameterized query and the array of parameters to the database server, which then performs the parameter binding and query execution.

The result can produce unparalleled performance, benefiting from a reduction in network traffic and distributing some of the workload to the server. The advantages of array DML are particularly beneficial for extract, transform, load (ETL) operations, such as those often associated with data warehousing.

Performance varies by database, as some databases natively support the operations performed by array DML. For the others, FireDAC emulates the operations performed by array DML, which might produce only marginal performance improvements. Databases that most benefit from array DML include InterBase, Firebird, IBM DB2, MS SQL Server, Oracle, and SQLite.

Array DML is demonstrated in the project named ArrayDML, whose main form is shown in the following figure.



This rather simple project employs Array DML to insert a large number of records into a database, creating one record for each calendar day over a range of dates. This task is performed by the following code:

```
procedure TForm1.Button1Click(Sender: TObject);  
var
```

```

    CurrentDate: TDateTime;
    TotalDays: Integer;
    i: Integer;
begin
    if DateTimePicker1.Date >= DateTimePicker2.Date then
        raise Exception.Create('Starting date cannot following ending date');

    TotalDays := DateUtils.DaysBetween(DateTimePicker1.Date,
                                        DateTimePicker2.Date);
    //Creating the table in a transaction that is committed
    //before beginning the INSERT process ensures that the
    //table is available when the Array DML process is initiated.
    FDConnection1.StartTransaction;
    try
        FDConnection1.ExecSQL('CREATE TABLE ' + Edit1.Text + ' ( ' +
                              '      DateValue date NOT NULL Primary Key, ' +
                              '      DayName varchar(16) ) ');
    finally
        FDConnection1.Commit;
    end;

    FDQuery1.SQL.Text := 'INSERT INTO ' + Edit1.Text +
                        ' (DateValue, DayName) VALUES (:Date, (:name) )';

    CurrentDate := DateTimePicker1.Date;
    //Set the array size
    FDQuery1.Params.ArraySize := TotalDays;
    //Insert the parameters
    for i := 0 to TotalDays - 1 do
    begin
        FDQuery1.Params[0].AsDateTimes[i] := DateOf(CurrentDate);
        FDQuery1.Params[1].AsStrings[i] := FormatDateTime('dddd', CurrentDate);
        CurrentDate := IncDay(CurrentDate);
    end;
    //Execute the query
    FDQuery1.Execute(FDQuery1.Params.ArraySize);

    //Let's take a look at the data
    FDQuery1.Open('SELECT * FROM ' + Edit1.Text);
    DataSource1.DataSet := FDQuery1;
end;

```

## SUPPORT FOR MULTITHREADING AND ASYNCHRONOUS QUERIES

FireDAC provides support for concurrent execution in a number of ways. First of all, FireDAC is thread-safe, so long as you ensure that connections and FireDAC TDataSets are used by only one thread at a time. For example, if you want to execute a lengthy stored procedure in a worker thread, that thread should have its own FDConnection, and the FDStoredProc or FDQuery component that you use to execute the stored procedure should use that connection and be executed entirely within the context of the thread.

Of course, any time you introduce additional threads of execution to your application, you need to rigorously follow sound multithreading practice, such as synchronizing access to resources shared by threads, and accessing objects created in the primary thread of execution from worker threads using the `TThread.Synchronize` method. Other prohibitions apply, but you get the picture.

FireDAC also support the asynchronous execution of queries. For example, if you set the `ResourceOptions.CmdExecMode` property of an `FDQuery` to `amNonBlocking`, FireDAC will execute the query on a worker thread. Importantly, FireDAC manages the details of this worker thread, destroying the thread once the query is finished. While this makes asynchronous execution easy to employ, you still need to observe sound multithreading practice. For example, a query executed asynchronously should not be connected to a `TDataSource` or `TBindSourceDB` during its execution. For example, you can call `FDQuery.DisableControls` to disassociate a query from its `TDataSource` prior to executing the query asynchronously, calling `FDQuery.EnableControls` from the `FDQuery.AfterOpen` event handler.

When a query is running asynchronously, you can monitor its progress by reading the query's `Command.State` property. Among other things, you can use this property to determine that the query is executing, is in the process of fetching the result set, or has completed execution and has an accessible result set.

Note: FireDAC permits only one query to be executed asynchronously at a time from your application's main thread of execution. If you need to execute two or more queries concurrently, you need to encapsulate those queries in separate instances of the `TThread` class.

There are several additional execution modes supported by FireDAC `TDataSets`. The default mode is `amBlocking`, which blocks the calling code until the query returns. The `amBlocking` mode is the recommended mode when executing a query asynchronously from a custom `TThread`. Using this mode, the calling thread is blocked until the query returns, but the all other threads, including the main thread (the user interface), remain responsive.

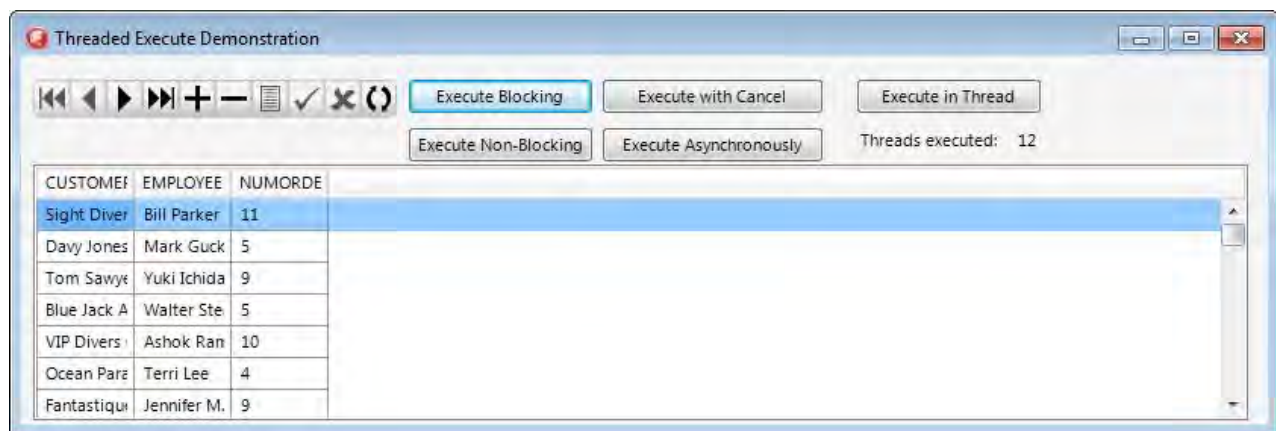
Execute `amAsync` executes the query asynchronously, like `amNonBlocking` (there is a difference between `amAsync` and `amNonBlocking`, but I am not entirely sure what that is). Finally, you can execute a query using the `amCancelDialog` mode. In this mode, a dialog box is displayed immediately prior to the execution of the thread. The query is then executed in a blocking mode, where as the calling thread and the user interface are blocked. Importantly, the user can interact with the dialog box, permitting the user to cancel (abort) the query before it returns. This mode is useful when you need to execute a



potentially lengthy query, but do not want to have to manage the additional complexity introduced by worker threads.

When your application includes more than one connection to a single database, something that will necessarily happen when you create custom worker threads from which to execute concurrent queries, you should consider using an `FDManager`. The `FDManager` class defines a connection definition that can be shared by multiple `FDConnections`, simplifying the process of configuring each `FDConnection` and potentially enabling connection pooling.

The use of `FDManager`, multiple `FDConnections`, blocking and non-blocking queries, as well as asynchronous queries, is demonstrated in the `ThreadedExecute` project, whose main form is shown in the following figure.



This project demonstrates the various kinds of query execution, including one that is encapsulated in a `TThread` instance. This project also demonstrates the use of an `FDManager` component to define a connection definition that supports a connection pool.

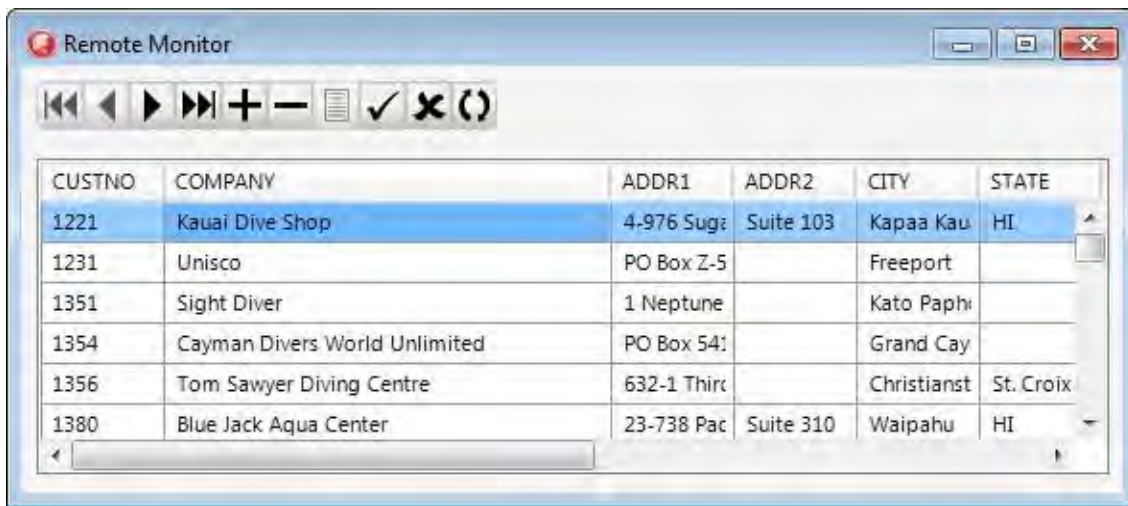
## POWERFUL MONITORING CAPABILITIES

FireDAC includes components and utilities that make it easy to trace FireDAC's interaction with the underlying databases. Using the `FDMoniFlatFileClientLink`, you can write trace information to a file, or you can implement your own monitor mechanism using the `FDMoniCustomClientLink`, which triggers an event handler with informative parameters. You implement your custom tracing operations from within this event handler.

But the real gem in this collection is the `FDMoniRemoteClientLink`. Using this component, you can monitor trace information at runtime using the `FDMonitor` utility that

ships with FireDAC. (The FDMonitor utility is located in RAD Studio's bin directory.) You can monitor local applications, but since FDMoniCustomClientLink and FDMonitor communicate via TCP/IP (transmission control protocol/Internet protocol), you can monitor the trace stream from almost anywhere on the Internet.

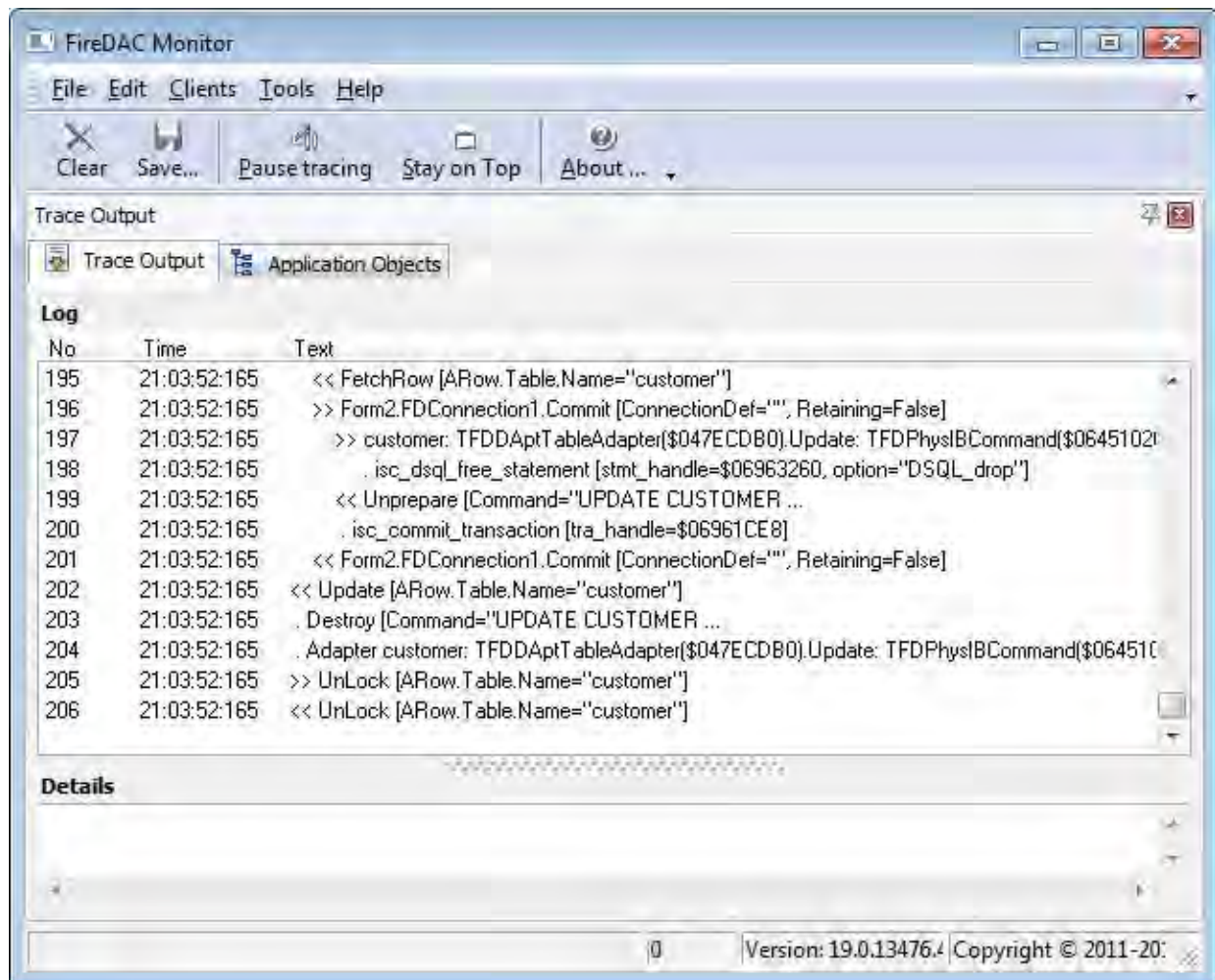
Remote monitoring is demonstrated by the RemoteMonitorFMX project, shown in the following figure.



The screenshot shows a window titled "Remote Monitor" with a toolbar containing navigation and control icons. Below the toolbar is a table with the following data:

CUSTNO	COMPANY	ADDR1	ADDR2	CITY	STATE
1221	Kauai Dive Shop	4-976 Sugz	Suite 103	Kapaa Kau	HI
1231	Unisco	PO Box Z-5		Freeport	
1351	Sight Diver	1 Neptune		Kato Paphi	
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cay	
1356	Tom Sawyer Diving Centre	632-1 Thirc		Christianst	St. Croix
1380	Blue Jack Aqua Center	23-738 Pac	Suite 310	Waipahu	HI

This project is a simple FireMonkey application that permits the editing of the customer table in the dbdemos.gdb InterBase database. So long as the FireDAC Monitor utility is running before this project is loaded, all SQL operations that FireDAC submits to the InterBase server can be viewed. In the following figure, a change has been made to the record shown in the preceding figure, and the monitor trace shown in the following figure represents FireDAC's updates when the Post button on the BindNavigator is clicked.



FDMoniRemoteClientLink has an additional capability that you might find useful. It permits you to dump the contents of the current record, or even the entire dataset, for retrieval by the FireDAC Monitor utility. In addition, custom objects in your application can be monitored, though this requires some custom programming. Note that you can also use CodeSite, a third-party utility from Raize Software that is bundled with RAD Studio projects to monitor object state.

## BUILT-IN DIALOG SUPPORT

One of the lesser, though certainly welcome, features of FireDAC is its predefined dialog boxes. FireDAC includes a handful of useful, specific purpose dialog boxes that can be added to your application simply by dropping a component onto your form or data module. For example, the FDGUIxLoginDialog can be used to automatically display a dialog box to capture a user's database username and password. Similarly, the FDGUIxErrorDialog can be used to display exceptions raised by FireDAC.

You use these dialog boxes simply by placing their corresponding component onto your form. FireDAC will then automatically display the associated dialog box when needed. For example, if you are attempting to connect to a database using the `FDConnection` component, and the `FDConnection.LoginPrompt` property is set to `True`, FireDAC will automatically display a login dialog box if the `FDGUIxLoginDialog` component is present.

Several of the projects on the code disk make use of build-in FireDAC dialog boxes. For example, the `ThreadedExecute` project employs a cancel query dialog box for use with executing a query using the `FDQuery.ResourceOptions.CmdExecMode` of `amCancelDialog`, as described in the section, *Support for Multithreading and Asynchronous Queries*. An `FDGUIxAsyncExecuteDialog` component was added to that project's data module, which forced the insertion of the necessary unit to the data module's uses clause, making the dialog box resource available to the project. Likewise, the `ClientServerFMX` project, described in the final section of this paper, uses a build-in login dialog box to capture the user's username and password.

## LOCAL SQL

I've saved my favorite feature for last, and that is local SQL. Local SQL permits you to execute SQL `SELECT`, `INSERT`, and `UPDATE` statements against any `TDataSet`. (Other DML statements may be allowed, but these are the obvious ones that most developers will be interested in.) For example, you can perform a query against an `FDTable` to gather simple aggregate statistics like `SUM` and `AVG` from the data it contains. Similarly, you can query an `FDQuery` and perform a left outer join to an `FDStoredProc` component (in which case the stored procedure must return a result set).

Importantly, this ability to query `TDataSets` is not limited to FireDAC `TDataSets`. As a result, there is nothing to prevent you from performing a query that performs a join between an `FDQuery`, a `SQLDataSet`, and a `ClientDataSet`.

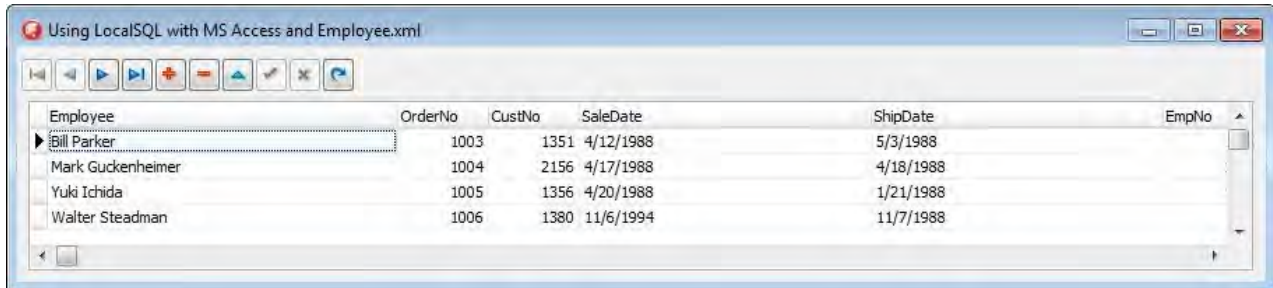
FireDAC performs this SQL slight-of-hand by using the SQL engine from the SQLite open source project. It's a very clever solution, and one that enables a whole range of interesting data-related solutions that would otherwise be difficult or nearly impossible to implement. Your joins don't have to be as complicated as I've described, but the benefits are obvious.

Note: The initial releases of some versions of RAD Studio were missing two files critical for working with SQLite. These files are included in the subsequent updates. If you encounter a compiler error indicating that one of these files is missing when building an application using the FireDAC SQLite driver, you can find the replacement files using the

following link. Place the file `sqlite3_x86.obj` in the `RAD Studio\lib\win32\release` directory, and the file `sqlite3_x64.obj` in the `RAD Studio\lib\win64\release` directory:

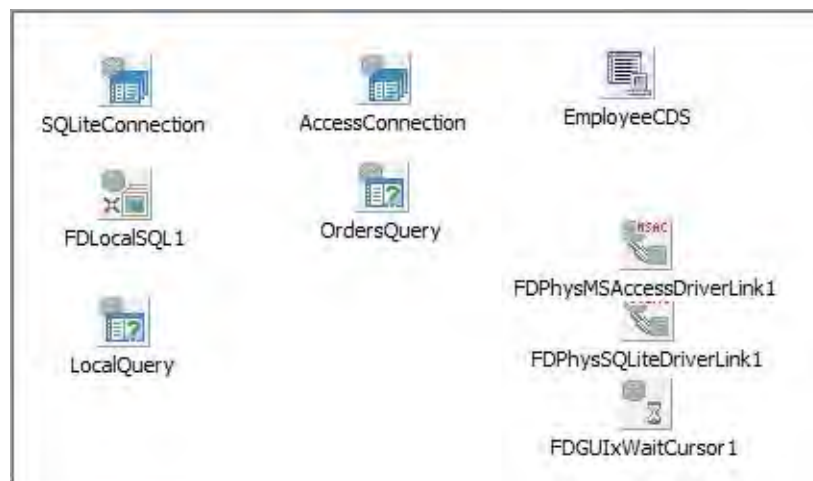
<https://forums.embarcadero.com/thread.jspa?messageID=600466&tstart=0>

The use of local SQL is demonstrated in the `AccessCDSLocalSQL` project, whose main form is shown in the following figure.



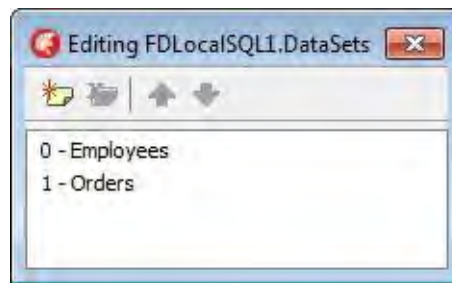
Employee	OrderNo	CustNo	SaleDate	ShipDate	EmpNo
Bill Parker	1003	1351	4/12/1988	5/3/1988	
Mark Guckenheimer	1004	2156	4/17/1988	4/18/1988	
Yuki Ichida	1005	1356	4/20/1988	1/21/1988	
Walter Steadman	1006	1380	11/6/1994	11/7/1988	

The data module from this project is shown in the following figure. Here you see the `SQLiteConnection` component, which simply loads the SQLite driver. The component named `FDLocalSQL1` identifies the `TDataSets` that can participate in the query, and the component named `LocalQuery`, which uses the `SQLiteConnection` as its connection, holds the actual query that is executed.



The following figure shows the `FDLocalSQL`'s `DataSets` property editor, which associates names with the `TDataSets` (`Orders` refers to the `OrdersQuery`, and `Employees` refers to `EmployeeCDS`).





The following is the query found in the LocalQuery component's SQL property:

```
SELECT e.FirstName || ' ' || e.LastName as Employee,  
       o.*  
FROM Orders o  
INNER JOIN Employees e ON e.EmpNo = o.EmpNo
```

## BUT THAT'S NOT ALL

I've outlined many of the major features of FireDAC, but there's more to FireDAC than what I've described so far. Here are some additional features of FireDAC that might interest you:

- FireDAC provides transparent connection recovery. When an FDConnection's ResourceOptions.AutoReconnect property is set to True, FireDAC will attempt to re-connect to a database when the connection is dropped. This feature is especially valuable in environments where the network connection is unstable or is not reliable.
- FireDAC supports a range of transaction options, including simultaneous and nested transactions.
- FireDAC makes it easy to use some of the supported database services, such as backup, restore, validation, SQLite custom function, and more. These services are accessed using the components from the FireDAC Services tab of the Tool Palette. Here you will find components that expose the underlying services for many of FireDAC's supported databases.
- FireDAC gives you tremendous control over how your data is accessed. FDManager, FDConnection, and FDCommand components have a collection of option-related properties (such as FetchOptions, ResourceOptions, and so forth) that each have many customizable properties. While the default options are normally well-suited for most data access, you have exceptional control over how FireDAC operates, permitting you to accommodate special cases. Furthermore, these options can be inherited, making it easy to customize these options on an application-by-application basis. For example, you can define the base options for



a connection definition using an `FDManager`, and these will be inherited by any `FDConnection` that use that connection definition.

- Just as FireDAC gives you a lot of control over connection options, it also permits you to customize the mapping of data types. You control the mapping of data types using the `FormatOptions` property of your `FDConnection`. If you set `FormatOptions.OwnMapRules` to `True`, you can define one or more `TFDMapRule` instances in the `FormatOptions.MapRules` property. Each `FDMapRule` describes to FireDAC a source data type and the data type to which it should convert the data in the destination. Data type mapping is especially valuable in applications that support two or more different database types, permitting you to map incompatibilities to a common set of data types.
- FireDAC supports the concept of virtual database drivers. A virtual database driver is a saved definition of a FireDAC driver, a specific database, and specific client library parameters. Saving these values in a connection definition file permits you to easily and quickly create new connections for the databases you use most often.
- FireDAC is written in the Delphi language. RAD Studio XE5 Professional includes some of the FireDAC source code, and the Enterprise, Ultimate, and Architect editions of RAD Studio, Delphi, and C++Builder include nearly all of the source code. This source code can help you understand how FireDAC works, and may even inspire you to create your own FireDAC database drivers.

For more information on these features, please refer to the FireDAC help in RAD Studio.

## HOW DO YOU GET FIREDAC?

A “local-only” edition of FireDAC is included in the Professional versions of RAD Studio XE5, Delphi XE5, and C++Builder XE5 (as well as in Enterprise, Ultimate, and Architect editions). This version is intended strictly for local and file server databases such as MS Access, dBASE, and Paradox. The Enterprise, Architect, and Ultimate editions of RAD Studio XE5, Delphi XE5, and C++Builder XE5 come with a wider set of FireDAC drivers, supporting local, client/server, and multitier development. Developers wishing to add client/server and multitier development options to their Professional edition can purchase the FireDAC Client/Server Add-On Pack from Embarcadero.

RAD Studio XE4 and related products included a license for FireDAC in their Enterprise and higher editions, and developers using RAD Studio 2010, Delphi 2010, C++Builder 2010, and later versions can use FireDAC by purchasing the FireDAC Client/Server Add-

On Pack. Using FireDAC in XE4 products required a separate installation, and it should be noted that while FireDAC will work with the 2010 and later products, it is officially supported only for XE4 and later versions.

FireDAC, as it appears in XE5, is different from the XE4 version in a number of ways. The most apparent of these differences is that the components have been updated to reflect their new branding (for example, the FireDAC connection component is now named `FDConnection`, compared to `ADConnection` in the XE4 version). Unit names and namespaces have been updated as well. These changes are only an issue if you are upgrading an existing FireDAC application from XE4 or earlier. As must be apparent by now, I am using the XE5 version in all of the examples shown in this whitepaper, and therefore only the updated component and unit names appear here.

## MIGRATING APPLICATIONS TO FIREDAC

This section details the process of migrating existing applications that currently use some other data access framework. Here I will approach four distinct topics. The first discusses how to bring your XE4 FireDAC and AnyDAC applications up to date with the new FireDAC class names and unit namespaces.

In the remaining three parts, I will talk about how to move existing applications that currently use some other data access mechanism to the FireDAC framework. In the first of these three parts, I will talk about how to transition an application that uses one of the other data access frameworks to FireDAC, specifically applications where the underlying database is one of those supported by one of FireDAC's native drivers. In the second section of these final three parts, I will discuss the migration of database applications for which there is not a native driver, and how you perform this migration using FireDAC's ODBC driver. This particular topic is essential for those of you still using the BDE, since your applications must be using an older file server database such as Paradox or dBASE and there is no native FireDAC driver for these databases.

The last part in this section discusses side-by-side deployment, the implementation of FireDAC in applications that will also continue to use another data access framework (at least for the near future). This section shows you how FireDAC can augment and support existing applications even without committing to a complete migration.

## MIGRATION OVERVIEW

The process of migrating an application to FireDAC entails implementing FireDAC data access in an existing application, either by completely replacing an existing data access mechanism, or by supplementing it.

Before getting to the topics in this section, let me address the topic of data access migration in general. First of all, as many of you know from my previous writings, I am a pragmatist. I take software development seriously, and I don't believe in change for the sake of change. An existing application that works properly is a thing of value, and there must be demonstrable benefits in order to justify the time and resources it takes to update, test, and re-deploy an application. As a result, I am not going to advocate for the migration to FireDAC of existing applications just because FireDAC is now available in RAD Studio. On the other hand, migrating to FireDAC typically results in an overall improvement in application performance, which might be justification enough to undertake the conversion.

Where I do advocate for the migration of existing applications to FireDAC are those situations in which either your existing data access framework cannot provide for the needs of your application or those situations where your application is destined to outlive the lifecycle of your existing data access framework. Clearly, applications that you are going to rely on going forward that still use the BDE fall into this category, and now is the time to plan and execute their migration. Likewise, if you have existing applications that use FireDAC XE4 or AnyDAC, those applications should be migrated as well, though in many instances, this migration requires little effort.

On the other hand, if your current data access framework appears to fulfill your current and anticipated future needs, does this mean that you should ignore FireDAC? No. The fact is, the use of FireDAC is not an all or nothing proposition, and FireDAC has many impressive capabilities that you might add to existing applications, or which you plan to incorporate during new development. I will discuss some of these in the upcoming section on side-by-side deployment.

There is one final point I want to make about migration, which is especially true when you are replacing an existing data access framework with FireDAC. The ease with which your applications can be migrated will rely in part on the architecture of your applications. Specifically, ease of migration is directly correlated to the extent to which you isolated the details of your data access within specific units and modules. For example, if all of your data access is implemented in one or a few data modules, migration will be easier

than if each form in your application includes TDataSets. In those cases, you can update the data module or data modules, and most or all of your work will be done.

By comparison, if TDataSet components appear throughout your application, for example, each form has one or more TTable, TQuery, or TStoredProc components, you will need to migrate every form in your application. Alternatively, you might take this opportunity to consolidate your data access to one or more data modules, streamlining your data connectivity and easing the migration.

Note: The following descriptions apply to Delphi applications. If you are migrating C++Builder applications, the steps will be different, but the general concepts will be similar.

## MIGRATING FIREDAC XE4 AND ANYDAC APPLICATIONS

If you created applications using FireDAC in its original Embarcadero XE4 release, or were a customer of DA-Soft's AnyDAC, you will want to migrate those applications to use the current version of FireDAC. Doing so will permit you to use the enhanced capabilities as FireDAC continues to evolve, as well as the new development options afforded by current versions of RAD Studio. For example, you won't have the option to pursue FireDAC Android development unless you migrate from the older versions. (In addition, support for the pre-XE4 AnyDAC is no longer available.)

The good news, as I alluded to in the introduction of this section, is that this process is straightforward. The latest version of FireDAC supports all of the classes that you might have used in your existing FireDAC/AnyDAC applications, though the names have changed and the units in which they are defined have changed as well. The new classes might support features previously unavailable, but there is practically 100% backwards compatibility, and unit names and class names constitute most of the changes that you'll need to make.

In short, the process requires two steps. You must change the names of your FireDAC classes from their original AD prefix (such as ADConnection and ADQuery) to their new FD prefix (FDConnection and FDQuery). This change is one-to-one, in that each ADxxxxx component has an FDxxxx counterpart.

Second, you must replace the old unit namespaces with the new unit namespaces. For example, in AnyDAC you will find the uADCompClient used in most uses clauses, while in FireDAC it is now named FireDAC.Comp.Client.

You can take one of two approaches: the automated approach or the manual approach. No matter which approach you use, however, it is mandatory that you backup your original source code before you do anything. I know that is obvious, but I have to say it anyway.

## THE AUTOMATED APPROACH

The automated approach makes use of reFind.exe, a utility that you will find in the RAD Studio bin directory. In most installations of XE5, you will find this utility in the following directory (for a 64-bit Windows installation):

```
C:\Program Files (x86)\Embarcadero\RAD Studio\12.0\bin
```

This utility uses the Perl regular expression engine to convert the contents of your .pas and other related files. The rules for conversion are found in a text file that you must pass as one of the arguments to your invocation of reFind. RAD Studio ships with example scripts for a Delphi conversion of a VCL application. You can find these script files in the following folder in RAD Studio XE5:

```
C:\Users\Public\Documents\RAD Studio\xx.x\Samples\Delphi\Database\FireDAC\Tool\reFind\AD2FDMigration
```

where xx.x is the version of RAD Studio that you are using, which in the case of RAD Studio XE5, is 12.0. If your applications were built using C++Builder, you should examine the scripts in the AD2FDMigration directory for clues as to how you can build your own scripts to perform the automated migration.

The RAD Studio XE5 DocWiki describes the process of converting XE4 FireDAC/AnyDAC applications to FireDAC at the following URL:

```
http://docwiki.embarcadero.com/RADStudio/XE5/en/Migrating\_AnyDAC\_Applications\_to\_FireDAC
```

Here you will find a couple of sample commands that you can adapt to perform the conversion using reFind from the command prompt. For example, to convert your unit names you are given the following command:

```
<RAD Studio>\Bin\reFind *.pas *.dpk *.dpr *.dproj *.inc /S /Y /I /W /B:0  
/X:<RAD Studio Demos>\Delphi\Database\FireDAC\Tool\reFind\  
AD2FDMigration\FireDAC_Rename_Units.txt
```

Note: This and the next two commands appear on more than one line due to their length. When entering these commands from the command prompt (cmd.exe), they must appear without any carriage returns.

In this command the tag <RAD Studio> refers to the directory in which RAD Studio is installed. Similarly, <RAD Studio Demos> refers to the location where the sample project files are located. I found that I had to make the following modifications in order to run reFind successfully:

```
"C:\Program Files (x86)\Embarcadero\RAD Studio\12.0\Bin\reFind"  
*.pas *.dpk *.dpr *.dproj *.inc /S /Y /I /W /B:0  
/X:"C:\Users\Public\Documents\RAD Studio\12.0\Samples\Delphi\Database\FireDAC  
\Tool\reFind\AD2FDMigration\FireDAC_Rename_Units.txt"
```

Similarly, to perform the conversion of class names, I used the following statement:

```
"C:\Program Files (x86)\Embarcadero\RAD Studio\12.0\Bin\reFind" *.pas *.dfm  
*.dpk *.dpr *.inc /S /Y /I /B:0 /X:"C:\Users\Public\Documents\RAD  
Studio\12.0\Samples\Delphi\Database\FireDAC\Tool\reFind\AD2FDMigration\FireDA  
C_Rename_API.txt"
```

You can find a detailed description of reFind at this URL:

```
http://docwiki.embarcadero.com/RADStudio/XE5/en/  
ReFind.exe,\_the\_Search\_and\_Replace\_Utility\_Using\_Perl\_RegEx\_Expressions
```

There is also a readme.txt file that introduces a sample batch file that you can adapt to automate your conversions, and this is particularly useful if you have the need to convert multiple applications. This readme.txt file can be found at the following location:

```
C:\Users\Public\Documents\RAD Studio\xx.x\Samples\Delphi\Database\  
FireDAC\Tool\reFind
```

where once again xx.x is the version of RAD Studio that you are using.

## THE MANUAL APPROACH

The manual approach is more involved, as it requires the opening of each module (data module, form, or frame) on which XE4 FireDAC and AnyDAC components reside, replacing the old components names and unit namespaces with the new names and namespaces. The manual technique is described in the following steps:

1. Backup the project.
2. Create a new project. If the project you are converting is a VCL application, create a new VCL application. If it was a FireMonkey application, create a new FireMonkey application.
3. Add a module on which FireDAC/AnyDAC components appear to the new project by selecting Project | Add to Project. (If this module has the same name as the



default form created when you created the new project, you must first remove that default form before performing this step.) You will see a warning that a component with one of the old names (i.e., TADConnection) cannot be found.

4. Select Ignore all. The module will be open and all of the older components will be removed. This module, in this state, cannot be compiled.
5. For each FireDAC/AnyDAC class that appeared on this module, place the corresponding FireDAC component. For example, if you previously had an ADConnection and an ADQuery, place an FDConnection and an FDQuery on that module.
6. Remove all of the old AnyDAC unit names from your modules uses clause. These names begin with the prefix uAD, such as uADStanIntf.
7. Save your project. At this point, you will get a message about each of your missing AnyDAC components, informing you that the declaration in the form does not have a corresponding component. Proceed by selecting to remove each of these missing components.
8. The next step is to make any code that referenced your AnyDAC component compilable. You can do this one of two ways. One is to rename each of your new FireDAC components, giving it the name of the corresponding AnyDAC component that was removed. For example, renaming FDQuery1 to **ADQuery1**. Better yet, find each instances of the old component name and replace it with the new component name. Once this is done, you should be able to compile your module. At this time (or at the time that you first save the changes to your module) the presence of the new FireDAC components will cause the new unit names to be added.
9. You are still not done. You must now configure each of the new FireDAC components that you have placed on your module. If the original configuration was complicated, you can refer to the original DFM or FMX files for a list of the properties and the values to which they were set.
10. Continue by repeating steps 3 through 9 for each of the remaining modules on which XE4 FireDAC or AnyDAC components appear.

## MIGRATING EXISTING APPLICATIONS TO FIREDAC

This section describes the process by which you can migrate an existing application from some other TDataSet implementation to FireDAC. This discussion assumes that the

legacy data access framework is currently installed and working properly, and that FireDAC has a native driver for the specific database. For example, if you are using dbExpress using the Microsoft SQL Server driver, and want to migrate your application to use FireDAC's Microsoft SQL Server driver, this section is for you. Likewise, if you have been using Paradox tables through the BDE, and are now migrating your application to an InterBase database and the FireDAC InterBase driver, you can also use this description.

However, if you are migrating a BDE Paradox application to FireDAC, and need to continue using Paradox tables, you should refer to the next section, *Migrating Applications Using ODBC*, which describes that process. The next section also applies to other databases for which there is no native FireDAC driver, but for which you have an ODBC driver.

There is at least one, and up to three, steps to migrating existing applications to FireDAC, where the existing application is currently using a non-FireDAC data access mechanism. The common step is the replacement of existing data access components with FireDAC data access components.

The other two steps may be optional, depending on the details of your migration. For instance, if you are migrating not only your data access framework, but also your database, you must take steps to transfer your data from the existing database to the new database. This can be accomplished in a number of ways.

Moving just the data can easily be accomplished with a `FDDataMove` component. This component takes any `TDataSet` as its source, and a FireDAC `TDataSet` as its destination. When you invoke its `Execute` method, it transfers data from the source dataset to the destination dataset. Alternatively, you can write your own data pump that reads your original data, typically one line at a time, and move the data into the new database using any of the available mechanisms for inserting records, such as `TFDDataset.CopyDataSet`, `TDataSet.Append`, `TDataSet.AppendRecord`, SQL INSERT queries, Array DML, and so forth.

Other aspects of the original database will probably need to be moved as well, such as stored procedures, constraints, views, user defined function, notifications, and so forth. This can be done manually, or in some cases you can write code or scripts that will perform the necessary actions.

The second step depends on the architecture of your legacy data access framework. For example, if you are currently using dbExpress, you are necessarily executing read-only queries. In order to edit that data, you are most likely moving that data into a

ClientDataSet and applying changes to the data there. FireDAC is much easier to use, since its TDataSets are typically editable.

How much time this step will take depends on what you want to accomplish during your migration. For example, in many cases you can simply replace your existing data access components with their FireDAC counterparts, and you are done. On the other hand, if you want to improve the usability of your user interface by taking advantage of FireDAC features, you will be committing to more work. In the case of moving from dbExpress to FireDAC, you might want to remove the ClientDataSets and permit the FireDAC TDataSets to write the data to the underlying database on a record-by-record basis. In a case like this, this step will take time, the amount of which will be directly proportional to the number of forms in your application and the complexity of your interface.

The actual process of swapping existing TDataSets for FireDAC TDataSets is pretty straightforward, but in most cases, you will proceed manually, placing FireDAC components on each module from which data access occurs. Furthermore, this process will be simpler to the extent to which you isolated your data access to one or a few data modules.

The manual process of replacing an existing data access framework with FireDAC is somewhat similar to that described in the preceding section about AnyDAC migration, but even easier. Specifically, when manually migrating from AnyDAC to FireDAC, you do not have both AnyDAC and FireDAC installed. However, when migrating from an existing framework to FireDAC, it is possible that you will have both frameworks installed at the same time. If this is the case, you can use the following steps. If the existing framework is not installed, you can proceed with the manual installation using the steps given for manual installation in the section, *Migrating FireDAC XE4 and AnyDAC Applications*. Here are the steps for migrating an existing framework to FireDAC, when you have both frameworks installed at the same time:

1. With both frameworks installed and available, open the backup of your existing application and display your first module (data module, form, or frame) from which data access occurs.
2. Add the necessary FireDAC components to this module and configure them for access to your database. For example, you might place and configure an FDConnection, after which you hook up an FDQuery and define its SQL SELECT statement.
3. Locate each of the components in your application that refer to the existing framework components and change their properties so that they now refer to the FireDAC counterparts. For example, if you are converting a data module on which

a ClientDataSet held data obtained through a dbExpress SQLDataSet, and have now implemented an FDQuery to hold that same data, locate any DataSource components that pointed to the ClientDataSet and change it to refer to your new FDQuery (or, locate the DataSetProvider used by your ClientDataSet and point it to the FDQuery). Similarly, locate any code that explicitly references the data access framework components that you are replacing and change that code to refer to the FireDAC instances you have added.

4. Once you have confirmed that your code compiles, and that your FireDAC components are providing the correct data, you can safely remove the old data access framework components and delete their corresponding units from your uses clauses.
5. Continue this process for any remaining modules from which data is accessed.

In those cases where your old data access framework implements an interface more complex than that required by FireDAC, you might want to initially hook the FireDAC components up in a manner similar to the framework you are replacing. For example, if you are moving from dbExpress to FireDAC, you might want to replace your SQLConnection and SQLDataSet components with FDConnection and FDQuery components, but keep the ClientDataSets and DataSetProviders in place. Doing so permits you to later update your user interface to use the FDQuery components instead of the ClientDataSets, as time and resources permit.

In some rare cases, it may be possible to use a technique like that described in the preceding part of this section where reFind was used to perform regular expression replacement of component names. Note, however, that using reFind to change the component classes from a non-FireDAC interface to a FireDAC interface is never as simple as the process of moving from AnyDAC to FireDAC. Even when there is a one-to-one correspondence between the old data access components and their FireDAC counterparts, there will be differences, some big and some small, between the property settings of these components. As a result, after a regular expression replacements the forms (DFM or FMX files) and source (.PAS) files, property assignments in the form files and programmatic assignments of properties in source files will not be valid, and will likely require extensive modifications.

## MIGRATING APPLICATIONS USING ODBC

This section describes migrating an existing application that uses a database for which there is no native FireDAC database driver. For example, migrating an application that

uses Paradox tables to FireDAC. There is no native Paradox driver in FireDAC. As a result, you will have to rely on the FireDAC ODBC driver, and utilize the Paradox ODBC driver supplied in Windows. This same approach applies to any migration where the database used by the application is not supported by a native FireDAC driver, but for which an ODBC driver is available.

Note: If your application supports dBASE or Visual FoxPro files, you might want to take a look at the FireDAC driver for the Advantage Database Server (and Advantage Local Server, a file server architecture). The FireDAC native driver for Advantage supports some dBASE formats, and you should determine whether or not you can use that native driver to connect to your dBASE files.

Before I continue, permit me to address a question that frequently comes up when I talk about migrating applications based on the BDE and Paradox tables to a new data access layer. Why, the question goes, would anyone want to continue using Paradox tables? If you are going to make the effort to migrate the application, isn't this the time to migrate the database as well, to a more stable, transaction supporting remote database server?

The question is a good one, and the answer is that migrating away from the inherently less stable file server based database format to a transaction-supporting remote database server is advisable, but not always possible. Here is an example that demonstrates what I am talking about. Imagine that many years ago, your company purchased an application that uses Paradox tables. Over the course of the years you have built one or more applications that supplement or extend that application, reading and possibly writing data to those same Paradox tables.

Now, after many years of use, that old application is still functioning well, or it is simply too complicated or costly to consider replacing. Unfortunately, the company from which that application was originally bought from no longer exists, or maybe it is just that the source code is unavailable. In any case, that original application will continue to be used and there are simply no acceptable options for changing the underlying database.

In a situation like this, your only option is to continue to support the original data format. And, in the case of Paradox (or dBASE) tables, your only options for moving away from the BDE involve migrating to an ODBC solution. And, as of RAD Studio XE4, the recommended ODBC supporting data access framework is FireDAC.

If the ODBC driver is not one that is available in all installations of Windows, you will also have to ensure that you provide for the installation of this driver as part of your application deployment. This can be accomplished by any number of third party

application installation programs, such as Inno Setup, a free installer utility that also has sample scripts for installing ODBC drivers.

There is an additional point that I want to make, and it applies specifically to the migration of BDE applications to FireDAC. FireDAC is an excellent option, compared to dbExpress, for migrating BDE applications, and no other data access mechanism in RAD Studio even comes close. The style of BDE development, which is often heavy on the "navigational" model, is supported beautifully by FireDAC. By comparison, the architecture of dbExpress often requires significant and time-consuming changes to the user interface and the approach to data access. FireDAC, quite simply, significantly reduces the resources required for migrating an application from the BDE. In most cases, once you've converted your TDataSets to FireDAC and established your connection to the database using ODBC, few or no additional changes are necessary. As a result, now is the time to plan your move from the BDE, relieving you from having to scramble when Embarcadero finally pulls the plug on the BDE. And this is expected to happen sooner than later.

Implementing FireDAC connectivity to your database using ODBC involves several steps. First, you configure your FDConnection to use the ODBC driver. Next, you define the parameters of this driver either by providing a reference to an existing ODBC data source name (DSN), or you provide all of the necessary connection information (typically associated with an ODBC connection string) the ODBCAdvanced parameter of the FDConnection.

There are a number of requirements before you can connect to your database using ODBC. Here is what is required:

- An installed ODBC driver for that database
- A FireDAC connection that uses the FireDAC ODBC bridging driver to connect to the ODBC driver
- The connection string parameters necessary to configure your ODBC driver to connect to your database

I am going to demonstrate this process by implementing ODBC access to the sample Paradox tables that ships with RAD Studio. I am choosing Paradox for several reasons. For one, Microsoft Windows ships with a Paradox ODBC driver, so it is something that should be available on your development machine. Second, Paradox table support is one of the reasons why there are so many RAD Studio developers still using the BDE. Since the BDE has been deprecated, I want to show you how easy it is to access Paradox tables from FireDAC. The third is that I have some of the connection string parameters for the



Windows Paradox ODBC driver, taken from <http://www.connectionstrings.com/microsoft-paradox-driver-odbc/>. These are shown here:

```
Driver={Microsoft Paradox Driver (*.db )};DriverID=538;Fil=Paradox 5.X;  
DefaultDir=c:\pathToDb\;Dbq=c:\pathToDb\;CollatingSequence=ASCII;
```

I am going to demonstrate two different ways to use an ODBC driver with FireDAC in this section. In the first, I am going to configure access to the ODBC driver using FireDAC's `FDConnection` component. In the second demonstration, I am going to show you how you can create a data source name for your ODBC driver, and how you can use that to connect to your database using FireDAC.

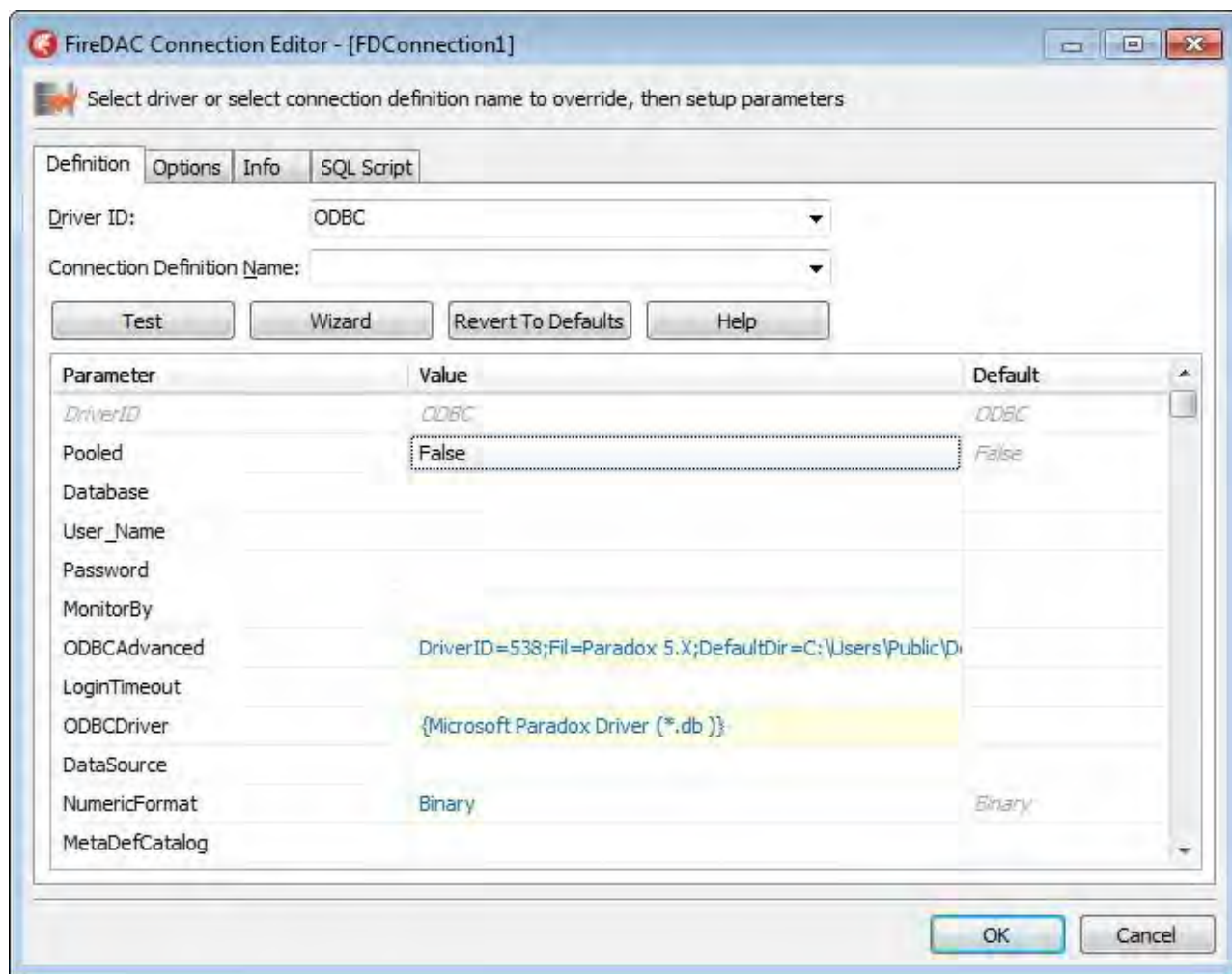
But before I start, I want to point out that the applications that I am creating use the 32-bit Windows ODBC driver for Paradox, which means that my application will be a 32-bit Windows application. If you need to build a 64-bit application, the steps will be slightly different, and you will also need a 64-bit ODBC driver for your database.

Use the following steps to create a connection and execute a query against a sample Paradox table that ships with RAD Studio:

1. Place an `FDConnection` and an `FDQuery` onto a data module.
2. Right-click the `FDConnection` and select `Connection Editor`.
3. Set the `Driver ID` drop down to `ODBC`. The FireDAC ODBC driver parameters appear in the `Connection Editor`.
4. Use the available drop-down list to set the `ODBCDriver` parameter to `{Microsoft Paradox Driver (*.db )}`. (Note, you include the matching curly braces, and there is exactly one space between the `'*.db'` and the `'}'`).
5. Set `ODBCAdvanced` to the following string. This string assumes that you are using RAD Studio XE5, in which the version is 12.0. If you are using a later version of FireDAC, replace 12.0 in both the `DefaultDir` and `Dbq` parameters with the corresponding version found in the sample directory path. Note, this string cannot include carriage returns or line feeds:

```
DriverID=538;Fil=Paradox 5.X;DefaultDir=C:\Users\Public\Documents\RAD Studio\12.0\  
Samples\Data\;Dbq=C:\Users\Public\Documents\RAD Studio\12.0\Samples\Data\  
CollatingSequence=ASCII;
```

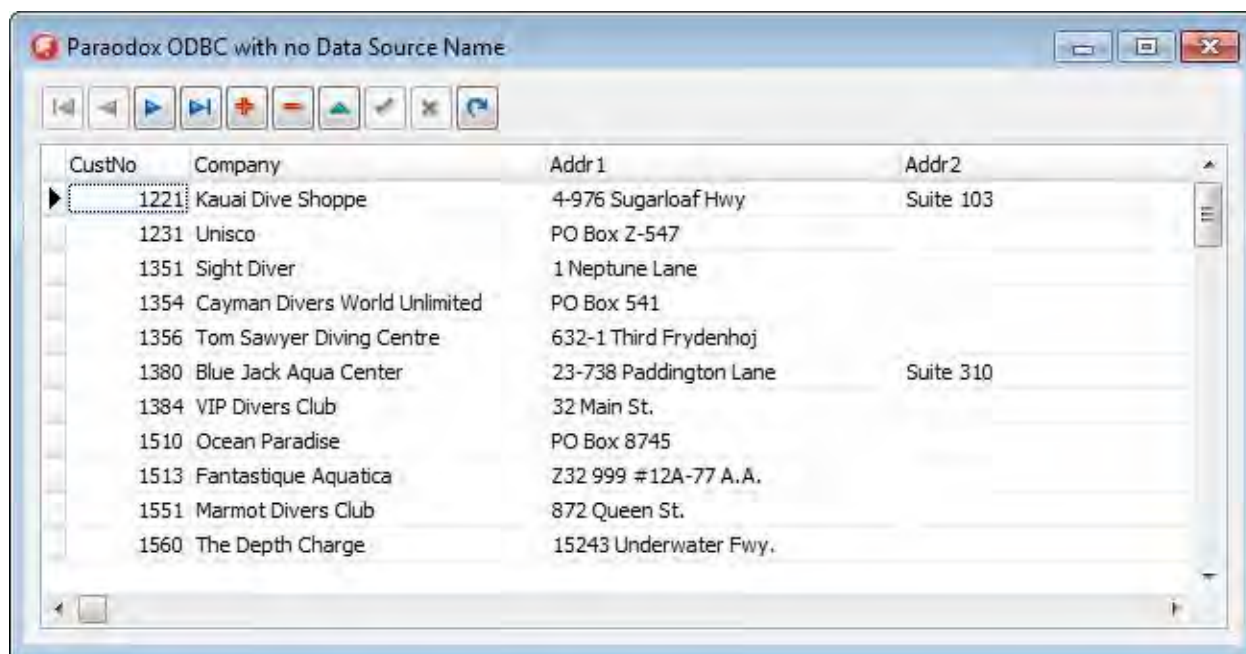
Your `Connection Editor` should now look something like that shown in the following figure.



6. Click Test. FireDAC will ask for a username, password, and other information. The sample database that ships with RAD Studio is not encrypted, so you can skip entering any information and click OK. A dialog box should confirm that you are connected. Close this dialog box and then save the Connection Editor by clicking OK.
7. Select the FDQuery and make sure that its Connection property is set to the FDConnection you just configured.
8. Set the FDQuery's SQL property to **SELECT \* FROM Customer.**
9. Next, set the FDQuery's Active property to **True.**
10. Finally, place one FDPhysODBCDriverLink and one FDGUIxWaitCursor onto your data module.

There, you are now connected to Paradox using FireDAC and the Windows Paradox ODBC driver. If you now use this data module from a form, and have a property

configured DBGrid, DBNavigator, and DataSource on this form, and the DataSource's DataSet property is set to the FDQuery on the data module, your form should look something like that shown in the following figure.



This project, named ParadoxODBCDynamic, can be found in the code samples.

My second example employs an ODBC data source name, or DSN. An ODBC DSN is a definition that you create on a workstation, after which it can be used to one or more applications to work with data through ODBC.

This project is similar to the ParadoxODBCDynamic project shown in the preceding figure. It differs in that the only parameters of the FDConnection Connection Editor that are set are the DriverID and DataSource properties.

Each time this project loads, it verifies that an appropriately named ODBC DSN exists. If so, it sets the Params property of the FDConnection. If the DSN does not exist, it attempts to create an appropriately configured system DSN. (A system DSN is visible to any user, while a user DSN is specific to only one of the workstation's users.)

This operation is shown in the following event handler, which is executed when the data module to which it is associated is created.

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);
var
    DataDir: string;
```

```

const
  DataSourceName = 'PDX Sample Data';
begin
  DataDir := 'C:\Users\Public\Documents\RAD Studio\' +
    FloatToStr(CompilerVersion - 14) +
    '\Samples\Data';
  if not ParadoxDSNExists(DataSourceName) then
    CreateParadoxDSN(DataSourceName, DataDir);
  FDConnection1.Params.Clear;
  FDConnection1.Params.Add('DataSource=' + DataSourceName);
  FDConnection1.Params.Add('DriverID=ODBC');
  FDQuery1.Open;
end;

```

This code defines a DSN of PDX Sample Data, and a data directory that points to the sample directory installed by RAD Studio. In this case, the version number is calculated by subtracting 14 from the CompilerVersion global variable, so this code should work correctly in most current versions of RAD Studio.

The ParadoxDSNExists and CreateParadoxDSN routines are found in a single separate unit, named ParadoxDSNx86. This is a custom unit that imports the SQLConfigDataSourceW Windows API function from ODBC32.DLL. ParadoxDSNExists uses the TRegistry class to determine whether or not the DSN exists, and CreateParadoxDSN to create the DSN if it is absent. The following is the entire text of then ParadoxDSNExists unit:

```

unit ParadoxDSNx86;

interface

uses Winapi.Windows, System.SysUtils;

function ParadoxDSNExists(DataSourceName: string): Boolean;

procedure CreateParadoxDSN(DataSourceName: string; DataDirectory: string);

implementation

uses Registry;

const
  ODBC_ADD_SYS_DSN = 4; // add a system DSN

function SQLConfigDataSource( hwndParent: LongWord ; fRequest: Word ;
  lpszDriver: PChar ; lpszAttributes: pchar ): boolean;
  stdcall; external 'ODBC32.DLL' name 'SQLConfigDataSourceW';

function ParadoxDSNExists(DataSourceName: string): Boolean;
var
  Registry: TRegistry;

```

```

begin
  Registry := TRegistry.Create;
  try
    Registry.RootKey := HKEY_LOCAL_MACHINE;
    Result := Registry.KeyExists('Software\Wow6432Node\ODBC\ODBC.INI\' +
                                DataSourceName);

  finally
    Registry.Free;
  end;
end;

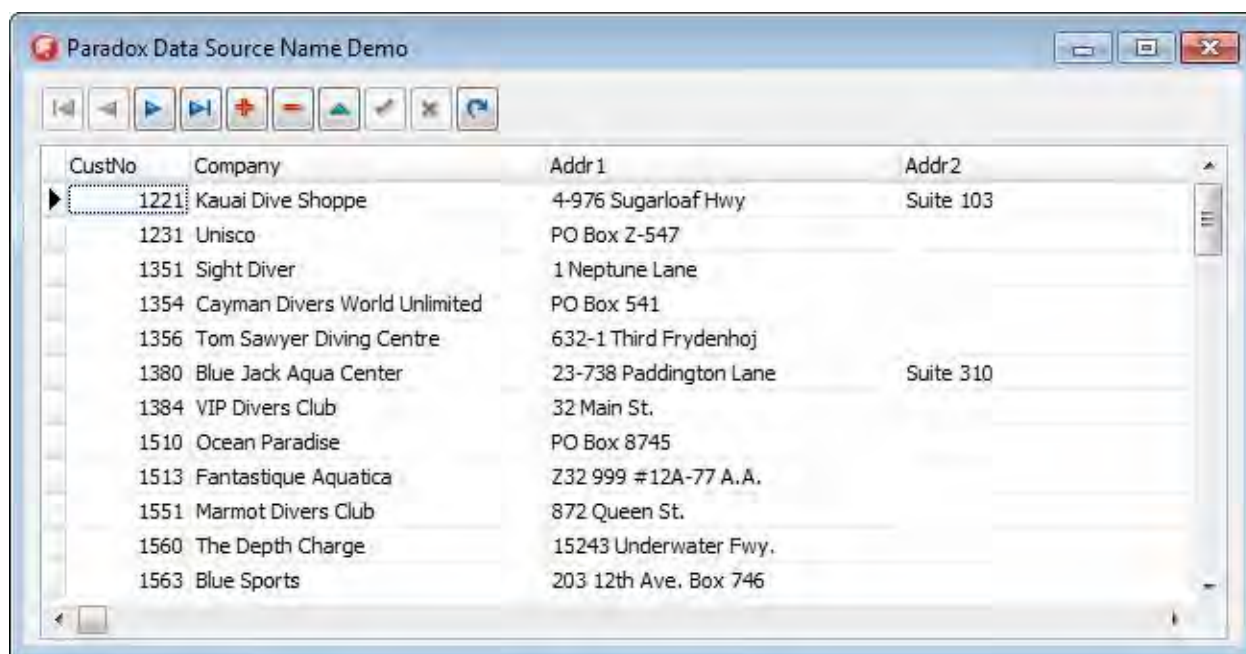
procedure CreateParadoxDSN(DataSourceName: string; DataDirectory: string);
var
  Attributes: string;
  RetVal: Boolean;
  DriverName: PChar;
  DirName: string;
begin
  DriverName := 'Microsoft Paradox Driver (*.db )';
  Attributes := 'DSN=' + DataSourceName + #0;
  Attributes := Attributes + 'DefaultDir=' + DataDirectory + #0;
  Attributes := Attributes + 'Dbq=' + DataDirectory + #0;
  Attributes := Attributes + 'Fil=Paradox 5.0'#0;
  Attributes := Attributes + 'DESCRIPTION=' + DataSourceName + #0#0;
  RetVal := SqlConfigDataSource(0, ODBC_ADD_SYS_DSN, DriverName,
                                PChar(Attributes));

  if not RetVal then
  begin
    Exception.Create('Failed to create data source name. Cannot continue');
  end;
end;

end.

```

There are two things to note in the preceding code. First, the value assigned to `DriverName` in the `CreateParadoxDSN` procedure must include the single space before the closing paren. Without this space, the driver name will not be recognized. Second, the data module that uses this unit must be created prior to the main form from which the `FDQuery` result set is consumed. The running main form of this project is shown in the following figure.



The screenshot shows a window titled "Paradox Data Source Name Demo". It contains a table with four columns: CustNo, Company, Addr1, and Addr2. The first row is selected, showing CustNo 1221, Company Kauai Dive Shoppe, Addr1 4-976 Sugarloaf Hwy, and Addr2 Suite 103. The table lists 14 rows of data.

CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box 2-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Fwy.	
1563	Blue Sports	203 12th Ave. Box 746	

This project, named ParadoxODBCDSN, can be found in the code samples.

FireDAC Architect Dmitry Arefiev, who provided the technical review of this paper, suggested that I remove this last project from this paper, arguing that creating the DSN on-the-fly was potentially error prone. I decided to retain this example, as I feel there are situations where creating a DSN for your ODBC driver is valuable. Nonetheless, if you use a technique similar to the one I showed, you should test it on a representative sample of machines to which the application will be deployed to ensure the success of your distribution.

Dmitry also suggested configuring FireDAC for an ODBC connection represented a good use case for creating a FireDAC virtual driver. A virtual driver is based on an existing FireDAC driver, but includes parameter definitions that point to a particular database and the other connection string data that the connection will need. The following code demonstrates how a virtual driver can be defined programmatically at runtime:

```
FDPhysODBCDriverLink1.DriverID := 'Paradox';
FDPhysODBCDriverLink1.ODBCDriver := '{Microsoft Paradox Driver (*.db )}';
FDPhysODBCDriverLink1.ODBCAdvanced := 'DriverID=538;Fil=Paradox 5.X;' +
    'CollatingSequence=ASCII;';

FDConnection1.Params.Add('DriverID=Paradox');
FDConnection1.Params.Add('Database= C:\Users\Public\Documents\' +
    'RAD Studio\12.0\Samples\Data\');
FDConnection1.Connected := True;
```



## SIDE-BY-SIDE DEPLOYMENT

As I mentioned at the outset of this section, it may not be practical to convert a legacy application to use FireDAC as the sole data access mechanism. But that doesn't mean that there is no place for FireDAC in those applications. While you might want to consider starting a new application using FireDAC alone, you should likewise consider adding FireDAC components as a supplement to existing applications based on another data access framework.

There are two features of FireDAC that immediately come to mind as offering benefits to any application, new or existing. These are local SQL and FDMemTable. As I described in the first section of this paper, local SQL permits you to execute SQL queries against any TDataSet, permitting you to perform client-side data manipulation and aggregation.

To use local SQL in an application, you will need to add at least several FireDAC components. At a minimum, you will need to add an FDConnection, an FDQuery, an FDLocalSQL, an FDPhysSQLiteDriverLink, and an FDGUIxWaitCursor to the project, most likely in a data module. The FDConnection defines a connection using the FireDAC SQLite driver, and the FDLocalSQL component points to each of the TDataSets that you want to include in the query. The FDQuery holds the actual query. Instead of table names, the query references the TDataSets that it is querying using their TComponent.Name, or an alias that you can optionally assign in your FDLocalSQL.DataSets property. Finally, the FDPhysSQLiteDriverLink and FDGUIxWaitCursor components ensure that the necessary resources are compiled into the project, though you can safely remove these two components once the appropriate units have been added to the uses clause (which will happen when the project is saved or compiled).

FDMemTables are similar to ClientDataSets, but offer significant performance benefits over their ClientDataSet cousins. An FDMemTable is not a replacement for a ClientDataSet, yet, but you might find situations where FDMemTable is preferable, and there is nothing stopping you from using it.

Who knows, after working with FireDAC in a hybrid application, you might decide that there are sufficient benefits to be gained by committing to a complete conversion to FireDAC for all of your application's data access needs. Whether that happens or not, it's good to know that FireDAC works well with other data access frameworks, and its features are at your disposal.

## FIREDAC IN ACTION

Over the course of this paper, I have introduced a number of applications that demonstrate a wide range of FireDAC capabilities. I conclude this paper with a look at five additional applications that include distributed and multi-device deployment of FireDAC applications.

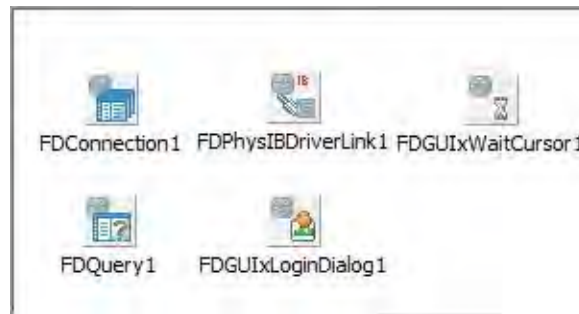
I begin with a simple client server application, moving next to a DataSnap server application built with FireDAC. The next two applications are DataSnap clients, one for Windows and another for use with mobile clients. While these clients are not technically FireDAC applications, they receive data that was accessed using FireDAC from the DataSnap server. The final application is a SQLite application designed for tablets.

## CLIENT/SERVER

This first project is named *InteBaseClientServer*, and it is the picture of simplicity. In some respects, it is nearly identical to the Microsoft Access application built from the steps provided in the section, *Making the Connection*. There is only one event handler in this entire application, and it is found in *OnCreate* event handler shown here:

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);  
var  
    DataDir: string;  
begin  
    DataDir := 'C:\Users\Public\Documents\RAD Studio\' +  
               FloatToStr(CompilerVersion - 14) +  
               '.0\Samples\Data\';  
    FDConnection1.Params.Clear;  
    FDConnection1.Params.Add('Database=' + DataDir + 'dbdemos.gdb');  
    FDConnection1.Params.Add('User_Name=sysdba');  
    FDConnection1.Params.Add('DriverID=IB');  
    FDQuery1.SQL.Text := 'SELECT * FROM Customer';  
    FDQuery1.Open;  
end;
```

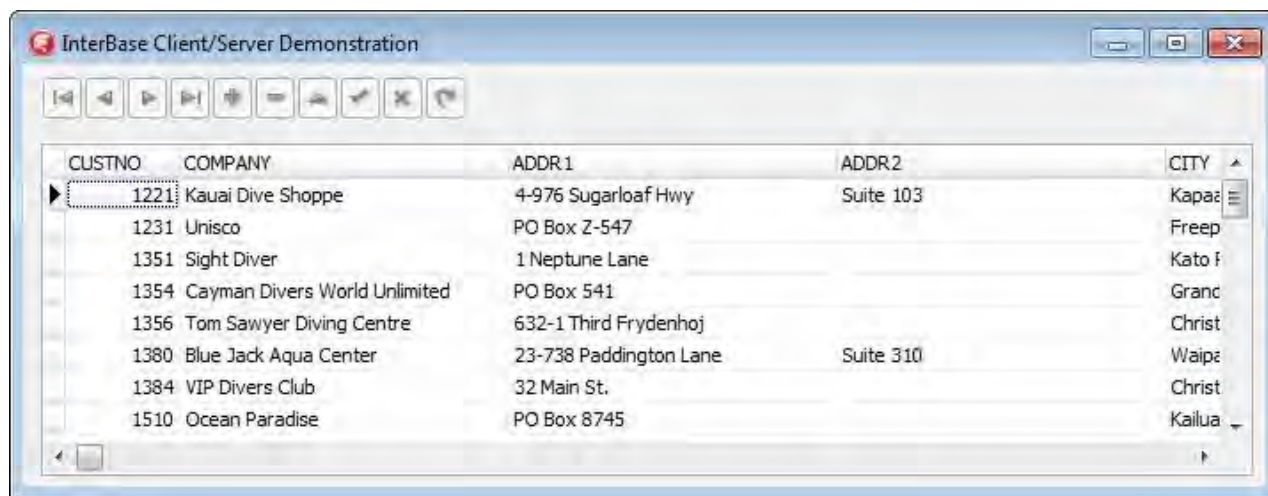
This project also demonstrates the use of an *FDGUIxLoginDialog* component, shown on the data module in the following figure.



The LoginPrompt property of the FDConnection is set to True, and the LoginDialog property is set to FDGUIxLoginDialog1. As a result, when the application runs, FireDAC automatically displays the login dialog, as shown in the following figure.



If the user types in the correct password (which is **masterkey** for this database), the application will run normally. If the user mistypes their password, they will see an exception and will be permitted to try again. Once they have successfully logged in, the main form is displayed, as shown in the following figure. From the main form, the user can insert records, delete records, and change data. Since CachedUpdates are not used by this application, any changes made by the user are written to the underlying database on a record-by-record basis.

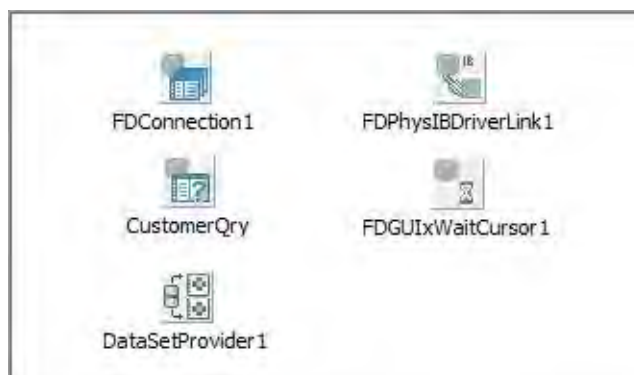


This application demonstrates just how simple it is to build your standard client/server applications using FireDAC.

## DATASNAP SERVER

The next application is a DataSnap server built using FireDAC. This particular server is a DBX server. A DBX server exposes TDataSets and custom methods via a data module that implements the IAppServerFastCall interface. This is the traditional API (application programming interface) originally introduced in MIDAS (IAppServer). I will use this DataSnap server to access the data retrieved by FireDAC from the DataSnap clients covered in the next section.

The key to the DBX server is the DSDataModule, shown in the following figure.

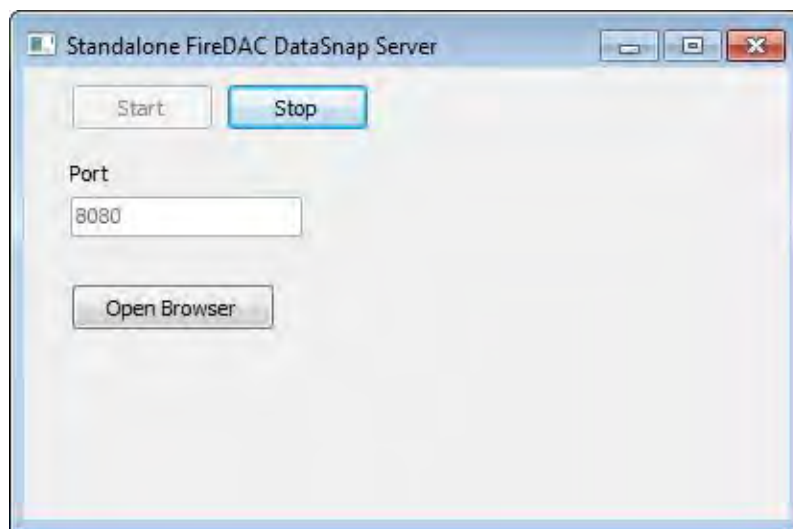


In many respects, this data module is identical to the one used by the InterBaseClientServer project shown in the previous section. The only differences here

are that the connection parameters include both the username and password, the `FDConnection` has its `LoginPrompt` property set to `False`, and no login dialog box is required. (Note that we could have implemented security for this server, but that is beyond the scope of this discussion.)

This particular DataSnap server was built as an HTTP server using `WebBroker`, and implemented as a standalone VCL application. This is a good way to build DataSnap servers, as it permits you to easily debug the server during development. In most cases, however, this server would be converted to an ISAPI `WebBroker` application, as that platform is best suited for deployment in the real world.

This particular server is listening on port 8080. The server, as a standalone VCL application, is shown in the following figure. As you can see in this figure, the server has been started, which is necessary in order for the DataSnap clients described in the following section to connect to the server.



There is much more to DataSnap than I have covered here. To learn more about DataSnap, and the latest features introduced to DataSnap in the most recent release of RAD Studio, please refer to the DataSnap-related presentations from CodeRage 8 or to the RAD Studio documentation.

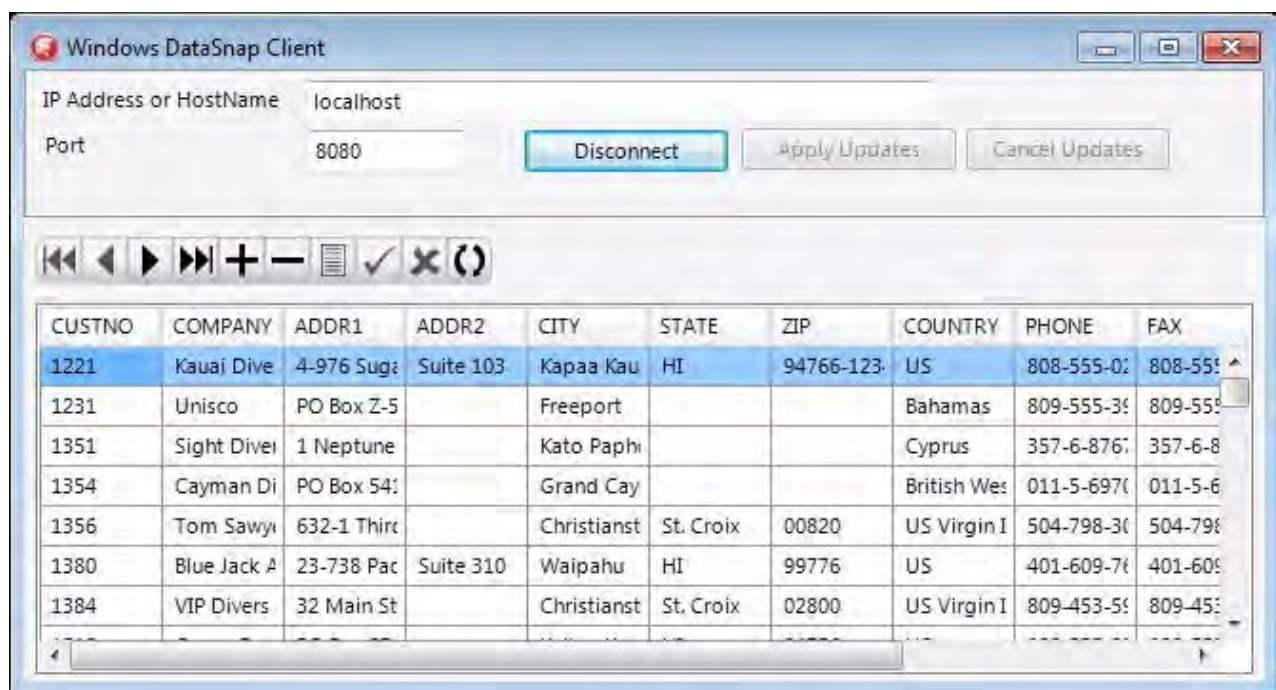
The code for this DataSnap server, and the following two DataSnap clients, can be found in the DataSnap folder in this paper's sample projects.

## DATASNAP CLIENTS

There are two DataSnap clients in this paper's sample projects. One is a Windows client and the other is a mobile client. Like the DataSnap server to which they attach, these are simple applications, presenting a single table for viewing and editing.

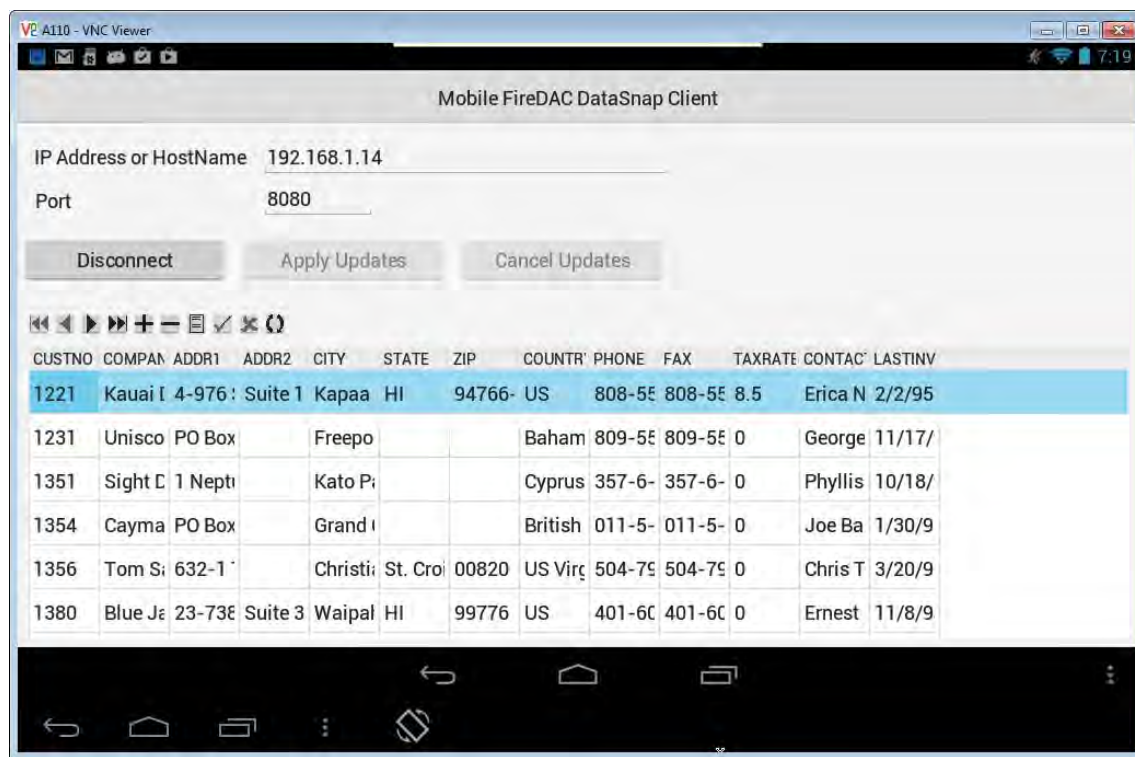
One feature of DataSnap clients is that they are typically thin clients, client applications that need no knowledge of databases, database drivers, or even the availability of a database client API. All of the data they receive is provided by the DataSnap server, which requires the client to know the IP address (or host name) of the server on which the DataSnap server is running, as well as the port upon which the server listens.

Here is the main form of the Windows DataSnap client, which is connected to the DataSnap server.



This Windows DataSnap client is using FireMonkey, RAD Studio's cross-platform component library. This same library, in fact, the very same FireMonkey components, were used to build the interface for the mobile DataSnap client. The mobile DataSnap client, compiled for Android, is shown running on a tablet in the following figure.



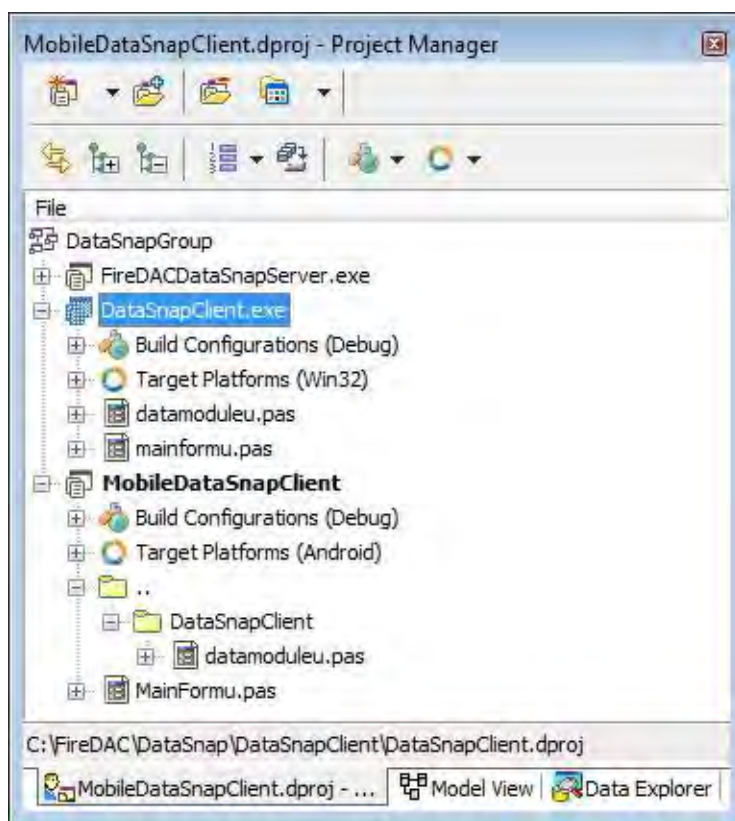


There are three points that I want to emphasize about these two DataSnap clients. The first is that they are both capable of viewing and editing the data that they are provided by the DataSnap server, from whichever platform they are deployed. (Even the Windows application could be deployed to OS X, although the user interface would be wrong.) All that is required is that the device on which they are deployed support network communications, and that the DataSnap server is running on a network that is visible to the client.

Changes that the user makes are cached locally by the client application until the user applies those changes back to the DataSnap server, by clicking the Apply Updates button. Alternatively the user can cancel any changes by clicking Cancel Updates. These buttons are only enabled when changes are present in the local cache.

The second point is that while the main forms of these two applications are slightly different in configuration (one is a FireMonkey desktop application and the other is a FireMonkey mobile application), these forms contain the same number of and types of FireMonkey components. In addition, these components are configured the same, and the underlying code is identical.

Finally, both of these clients compile with a single, common data module. This can be seen in the Project Manager shown in the following figure.



Taken together, these features serve to emphasize the fact that RAD Studio supports native compilation to multiple platforms and multiple devices from a common code base.

While I am at it, I want to acknowledge that the interface for the mobile application is not ideal. Mobile applications have different user interface requirements, and my use of the FireMonkey Grid is not traditional. For a more detailed discussion of mobile user interface development, you should consider viewing the CodeRage 8 presentation, *Designing Common User Interfaces for iOS & Android*, by Sarina DuPont.

You might also argue that these DataSnap client applications are not "true" FireDAC applications. While they obtain their data from a DataSnap server that uses FireDAC, there are no FireDAC components in either of these DataSnap clients. Instead they rely on the SQLConnection component from dbExpress in addition to a ClientDataSet. While I could have used FireDAC components in these clients, enabling both reading and writing to the DataSnap server, which would have required code that was specific to this database (no such dependencies are present in these clients). As RAD Studio moves forward, we are likely to see an emphasis on FireDAC in the DataSnap client, and additional options for enabling easy-to-create read/write clients. These future improvements were alluded to by Delphi Product Manager Marco Cantú in his CodeRage 8 session entitled, *"DataSnap Architectures, Optimizations, and Use Cases"*.

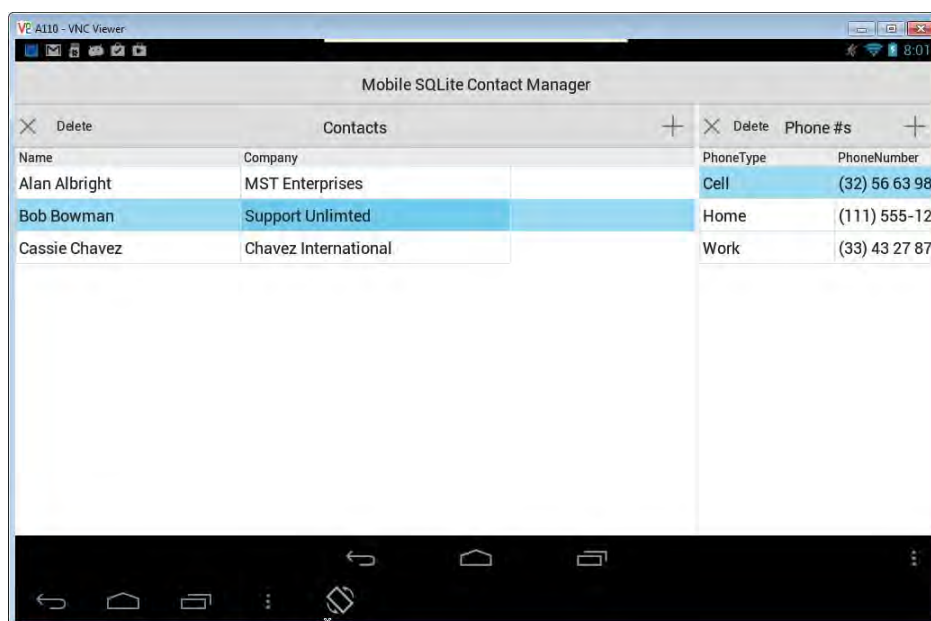
## SQLITE CLIENT FOR MOBILE

The final application makes use of FireDAC's ability to use local databases on supported platforms. This application, like the two DataSnap clients, employs the FireMonkey cross-platform component library. In addition, it uses SQLite, an open-source database that is available for all of RAD Studio's supported platforms.

The first time this application runs on the deployed platform, it creates the database it uses in the platform's user documents location. From that point forward, it uses this database to store a simple contact application that includes a master-detail relationship between individual contacts and their various phone numbers.

Note: While developing, each time you deploy to your Android device, RAD Studio uninstalls the previous installation. Since the SQLite database was placed in the user documents location, RAD Studio also uninstalls the database (which will be re-created the next time the application is run). This is a detail, but it might be important to you if you do not want your database removed each time you re-deploy your application to a particular device.

The following figure shows this application running on an Android tablet.



Once again, the use of FireDAC and FireMonkey has produced an application that can be deployed to multiple platforms with virtually no change to the underlying code. In reality, however, it is likely that you will use common code to implement the internals of your

applications, and design your user interfaces in a manner appropriate for the platforms to which they will be deployed. For example, this application implements the bulk of its functionality in a data module that would be suitable for any target platform. As a result, that same code base could easily be reused by two or more applications designed for different target platforms.

## SUMMARY

FireDAC is RAD Studio's latest data access mechanism, and in the short time since its release it has become the data access framework of choice for RAD Studio developers. FireDAC is cross-platform, compiles native code for multiple devices, supports more features than any previous data access mechanism in RAD Studio, and is written entirely in Delphi. While you can add FireDAC features to your legacy applications now, you should consider using FireDAC as your principle data access framework for new development.

## APPENDIX A: THE TDataSet INTERFACE

TDataSet is an abstract class declared in the Data.DB unit, and it lies at the core of RAD Studio's support for data access. This central role is emphasized by the fact that all of the visual component libraries (VCLs) data-aware controls, as well as those of nearly all third-party vendors, connect to data by means of a TDataSource. The TDataSource obtains this data from the object assigned to its DataSet property, whose type is TDataSet.

FireMonkey components, by comparison, do not include a specific collection of data-aware controls since most visual controls can be bound to data using LiveBindings. Nonetheless, FireMonkey components are bound to data through the BindSourceDB class. Like the DataSource class, BindSourceDB includes a DataSet property, which again is of type TDataSet.

Since Delphi 6, Delphi, and subsequently RAD Studio products, have shipped with five distinct implementations of TDataSet, and a number of third-party vendors provide additional implementations. Beginning with RAD Studio XE4, a sixth TDataSet implementation, FireDAC, was added. As a result, the six that ship with RAD Studio XE5 are associated with FireDAC, the Borland Database Engine (BDE), dbExpress, dbGo (ADO), MyBase (TClientDataSet), and IBExpress (InterBase Express). Examples of third-party TDataSet implementations include the Advantage Delphi Components from Sybase (an SAP Company), DOA (Direct ORACLE Access) from Allround Automations, and Pervasive Direct Access Components (PDAC) from Pervasive Software.

There are a number of ways to look at TDataSet components. While many of them provide you with access to a table-like structure of rows and columns, some of them do not (or support some data-related operations that do not return a result set). Furthermore, while some of them support the full range of the methods and properties introduced in the TDataSet class, there are quite a few TDataSet descendants that support a relatively small subset of the TDataSet methods and properties (in most cases, raise an exception when non-supported methods and properties are used).

Both of these characteristics are interesting in their own right, and are discussed in the following section.

## RESULT SET OR NO RESULT SET

When properly configured, many types of TDataSet instances return a result set when their Active property is set to True or their Open method is called. Examples of these types of components include FDTTable, TTable, TSQLTable, TADOTable, and TClientDataSet. These components can be seen in the following figure.



For these components, another side effect of becoming active is that they expose metadata about the structure of the result set. Metadata, which is data about the data, such as data type, precision, name, and so forth, can be accessed through TFields that are exposed by the TDataSets.

While all of the above mentioned TDataSets always expose metadata when they become active, they may not actually include data. For example, if an FDQuery selects from an InterBase table for which no records match the predicates in the WHERE clause, the result set will be empty. In that case, the metadata will reflect the structure of the selected values, but there will be no data to read.

When the result set contains data, these TDataSets, once active, will point to one of the records in the returned result set, specifically, the first record in the result set. The record to which the TDataSet points is referred to as *the current* record. Methods of the exposed TFields can then be used to read from, and in some cases write to, the current record.

With these types of TDataSets, you can use methods inherited from TDataSet to change which record is the current record, permitting you to read (and sometimes write) to a different record. Examples of these methods include Next (move to the next record in the result set) and Last (move to the last record in the result set). In addition, you can use the EOF property (end of file) to test whether the last call to Next attempted to move beyond the end of the result set, which signals that there are no more records in the result set.

As mentioned at the beginning of this appendix, not all TDataSets return a result set under every situation. For example, depending on how they are being used, the following TDataSets may or may not return a result set: TFDQuery, TQuery, TSQLStoredProc, TADOQuery, and TIBStoredProc. These components are shown in the following figure.





Whether one of these components returns a result set or not depends on the query being executed or the stored procedure being invoked. For example, if you are executing a data definition language (DDL) SQL statement, such as a CREATE TABLE statement, the query does not return a result set. Likewise, if you are using a stored procedure component, like TFDStoredProc, to execute a stored procedure that performs a task on your remote database, but which does not return data, no result set will be available.

What makes this distinction so important is that you cannot use the Active property or Open method on these TDataSets when they do not return a result set (doing so raises an exception). Instead, you must call a method such as ExecSQL (for query components) and ExecProc (for stored procedure components). (In reality, the method that you call to execute one of these components when it does not return a result set may be something other than ExecSQL or ExecProc, since these methods are not introduced in a common ancestor. The actual method name therefore relies on the specific class of TDataSet that you are using. Note also that FireDAC TDataSets provide the OpenOrExecute method, which works on both TFDDataSet that return result sets as well as those that do not.)

## COMPONENTS THAT SUPPORT TDATASET AND THOSE THAT DO NOT

Some of the TDataSet instances support all of the methods and properties of the TDataSet interface, but there are a significant number that do not. Furthermore, the same component may permit certain methods or properties to be used under some conditions, but not others. Why this is the case is explained in this section.

Let's consider the second item first, that a given TDataSet will permit you to use certain methods and properties under some conditions but not others. Actually, this was implied in the preceding section. Specifically, a TFDQuery that defines a query that returns a result set permits you to set the Active property to True, as well as call the Open method. On the other hand, a TFDQuery that executes a query that does not return the result set raises an exception if you call Open (or set Active to True).

Even if you accept that most TDataSets support some of the TDataSet methods conditionally, it is still a fact that some TDataSets, such as TFDTable and TClientDataSet,

support most (if not all) of the TDataSet methods most of the time, while others, such as TSQLDataSet, support a relatively small subset of the TDataSet interface. To understand why, we need to return to 1999 and the release of Delphi 5.

Delphi 5 was the first version of Delphi to introduce a data access mechanism that did not employ the BDE. That mechanism, which was referred to as ADO (ActiveX Data Objects) at the time (now referred to by the virtually meaningless name dbGo) introduced some concepts that did not match the design patterns used by the BDE.

But things got really interesting with the release of Kylix and the subsequent release of Delphi 6. This is when dbExpress entered the picture. The most significant feature of the dbExpress components, as far as this discussion is concerned, is that when one of the dbExpress TDataSets returns a result set, it is by definition a forward-only navigating (unidirectional), readonly cursor to the data. If you want to view this data in a data-aware control using the TDataSet interface, you needed to first load that data into a TClientDataSet, typically by using a TDataSetProvider. In conjunction with a TDataSetProvider, any changes made to that data can ultimately be written back to the database from which it was retrieved.

Since the SQLDataSet, SQLQuery, and other similar components of dbExpress were TDataSet descendants, they necessarily inherited methods of TDataSet that were completely incompatible with their nature. These included methods such as Edit, Prior, First, Post, as well as properties such as RecNo in a write mode.

If you think about it, the team building the VCL had to make a decision. They could have modified the TDataSet interface, removing those methods and properties that cannot apply to all descendants, or live with the interface that was introduced with the original Delphi. Did they make the right decision?

Having watched the VCL team since the earliest days of Delphi, and being a great admirer of their culture of excellence (with respect to object-oriented design), I cannot help but think they did the right thing. Sure, the TDataSet interface does not satisfy the current needs of all TDataSets, but interfaces, once published, should not be messed with. (Of course, we could debate this issue all day long, and there would be intelligent people on both sides of the issue, but let's agree that an interface, even if exposed through an abstract base class, is a contract, and we have to agree that the VCL team honored the TDataSet contract.)

Whether you agree with the VCL team's decision or not, the result is that there are TDataSet descendants for whom calling certain methods, or trying to use certain properties, will cause exceptions to be raised. As a result, it is your responsibility, when

using these classes, to ensure that you are using them the way they were intended, regardless of what Code Insight suggests you can do with them.

## TFIELDS AND TPARAMS

In the remainder of this appendix, I am going to focus on the mechanisms used to access the data of result sets (when the TDataSet returns a result set) and the parameters of queries and stored procedures. TFields are objects that represent the individual columns (or fields) of a TDataSet when it returns a result set. TParams, or similarly named classes, permit you to define the values of query parameters and stored procedure input parameters, and sometimes the output parameters of an executed stored procedure.

### ACCESSING A TDATASET'S DATA USING TFIELDS

With very few exceptions, when a TDataSet returns a result set, the individual fields of the result set can be read using TFields exposed by the TDataSet. In addition, if the result set is modifiable, individual fields can be modified using TFields. In addition to this role, TFields also expose the metadata of the underlying columns of the result set.

For TDataSets that expose their data through TFields, there are a number of ways to access the individual TFields, and this typically is done either by using the ordinal position of the TField in the TDataSet's structure or the underlying column name.

To access a TField by ordinal position, you can use the Fields property of the TDataSet, indexing it with the zero-based position of the underlying column in the TDataSet's structure. For example, if you know that the first field in a TDataSet named CustomerQry is an integer field, you can use the following statement to read, and conditionally write, the data in that field:

```
if CustomerQry.Fields[0].AsInteger = 0 then  
begin  
    CustomerQry.Edit;  
    CustomerQry.Fields[0].AsInteger := 1;  
    CustomerQry.Post;  
end;
```

If you want to access a field by the name of the field, you have two methods at your disposal. You can call FieldByName or FindField. Both of these methods take a single string parameter to which you pass the name of the field you want to access. If the value you pass to FieldByName corresponds to the name of a field in the TDataSet, it returns a

reference to the underlying TField. If the value is not the name of a field in the TDataSet, FieldByName raises an exception. The use of FieldByName is shown in the following code segment:

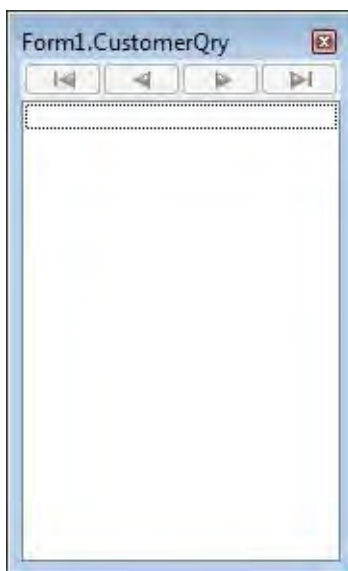
```
if CustomerQry.FieldByName('Status').AsInteger = 0 then  
begin  
    CustomerQry.Edit;  
    CustomerQry.FieldByName('Status').AsInteger := 1;  
    CustomerQry.Post;  
end;
```

By comparison, FindField never raises an exception. Instead, it returns a TField when the string parameter resolves to the name of a column in the TDataSet result set, otherwise it returns null (and you should test to ensure that it does not return null before attempting to read or write to the associated TField). Technically, FindField is a method of the object exposed in the Fields property of the TDataSet. However, since Fields is the default property for TDataSet, you can use TDataSet.FindField to execute this method instead of the more wordy TDataSet.Fields.FindField.

While FieldByName and FindField often make your code more readable, by making it clear in your code which field, by name, you want to access, both of these methods are more expensive, performance-wise, to call compared to using the Fields property. Using the Fields property, the index you use points directly to the underlying TField. By comparison, each and every time you call FieldByName or FindField, the Fields collection of the TDataSet is searched, field by field, for a field whose FieldName property matches the parameter you passed to these methods.

There is another technique for referring to TFields that does not even require a call to FieldByName or FindField. This technique is available when the result set that your TDataSet points to can be defined at design time, and it involves creating persistent TFields for your TDataSet.

If it is possible for you to define your result set at design time, you can use the Fields Editor component editor to create persistent fields at design time. For example, imagine that your TDataSet returns a result set that contains fields such as Status, AccountNumber, FirstName, LastName, and so forth. If you can access that result set at design time, you can right-click the TDataSet and select Fields Editor from the context menu. This displays the Fields Editor shown in the following figure.



From the Fields Editor, right-click and select Add All Fields. The Fields Editor will now look like that shown in the following figure, where one persistent TField has been created for each corresponding column in the underlying result set.



Once the persistent fields have been added, you will also notice corresponding TField descendants in the published section of your TForm declaration. The following is a portion of a TForm type declaration that shows the TField descendants that were created using the Fields Editor:

```
type
  TForm1 = class(TForm)
    CustomerQry: TFDQuery;
```

```
CustomerQryStatus: TIntegerField;  
CustomerQryAccountNumber: TStringField;  
CustomerQryFirstName: TStringField;  
CustomerQryLastName: TStringField;  
CustomerQryAddress1: TStringField;  
CustomerQryAddress2: TStringField;  
CustomerQryCity: TStringField;  
CustomerQryState_Province: TStringField;  
CustomerQryCountry: TStringField;  
CustomerQryPostalCode: TStringField;  
CustomerQryDateOfBirth: TDateField;  
CustomerQryCreditLimit: TIntegerField;  
//...
```

When the TDataSet with which these TField descendants are associated becomes active, these TFields point to their corresponding columns in the TDataSet result set. As a result, the following code could be used to produce the same effect as that produced by the two code segments shown earlier:

```
if CustomerQryStatus.AsInteger = 0 then  
begin  
    CustomerQry.Edit;  
    CustomerQryStatus.AsInteger := 1;  
    CustomerQry.Post;  
end;
```

## DEFINING PARAMETERS USING TPARAMS

Most of the TDataSets designed to execute queries or stored procedures support classes that you can use to define parameters for parameterized queries, as well as input and output parameters of stored procedures. These classes often have names like TParams (TFDParams in FireDAC), and similar to TFields, they can be accessed using either a property or through methods.

The following code demonstrates how to define a parameterized query, use the Params property to define the single string parameter of this query, followed by execution of the query. The parameter in this query is named accountno, and it is preceded by a colon, which in this particular database is used to identify a named parameter:

```
CustomerQry.SQL.Text :=  
    'SELECT * FROM CUSTOMER WHERE AccountNumber = :accountno;  
CustomerQry.Params[0].AsString := 'N00281';  
CustomerQry.Open;
```

To access the parameter by name, you can use the ParamByName method, as shown in the following code:



```
CustomerQry.SQL.Text :=  
  'SELECT * FROM CUSTOMER WHERE AccountNumber = :accountno;  
CustomerQry.ParamByName('accountno').AsString := 'N00281';  
CustomerQry.Open;
```

There is more to the TDataSet interface than demonstrated here. For more information on the TDataSet interface, database development in general, and FireDAC specifically, please refer to the available documentation at the following URL:

[http://docwiki.embarcadero.com/RADStudio/XE5/en/Developing\\_Database\\_Applications](http://docwiki.embarcadero.com/RADStudio/XE5/en/Developing_Database_Applications)

## ACKNOWLEDGMENTS

I want to express my thanks for the generous help of the many people who were involved in the production of this white paper and the associated Webinar. In particular, I want to thank Dmitry Arefiev, FireDAC Architect, for his astute technical review of the first draft, and subsequent technical support. I want to also thank Tim Del Chiaro, Embarcadero Technologies, and Loy Anderson, President, Jensen Data Systems, Inc., for their copyediting and proofreading assistance. Also, I want to thank Steve Haney, Senior Director of Marketing, Embarcadero Technologies, for guiding this project from its inception. Finally, I want to thank the following people at Embarcadero Technologies for the support, time, and direction in the execution of this paper and the corresponding Webinar: David Intersimone (David I), John Thomas (JT), Marco Cantú, Jason Vokes, Jim McKeeth and Tim Del Chiaro.

## ABOUT THE AUTHOR

Cary Jensen is Chief Technology Officer of Jensen Data Systems. Since 1988, he has built and deployed database applications in a wide range of industries. Cary is a developer, a consultant, an Embarcadero MVP, a bestselling author of more than 20 books on software development, and holds a Ph.D. in Human Factors Psychology, specializing in human-computer interaction. He and fellow Embarcadero MVP Bob (Dr.Bob) Swart present the annual Delphi Developer Days tours. Visit <http://DelphiDeveloperDays.com> for more information. Cary's latest book, Delphi in Depth: ClientDataSets, is available from <http://www.JensenDataSystems.com/cdsbook>.

## ABOUT EMBARCADERO TECHNOLOGIES

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance, and accelerate innovation. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. [www.embarcadero.com](http://www.embarcadero.com).