DataSnap Development

By Dan Miser

CHAPTER

21

IN THIS CHAPTER

Multitier applications are being talked about as much as any topic in computer programming today. This is happening for good reason. Multitier applications hold many advantages over the more traditional client/server applications. Borland's DataSnap is one way to help you create and deliver a multitier application using Delphi, building on techniques and skills you've accumulated when using Delphi. This chapter walks you through some general information about multitier application design and shows you how to apply those principles to create solid DataSnap applications.

Mechanics of Creating a Multitier Application

Because we'll be talking about a multitier application, it might be helpful to first provide a frame of reference to what a tier really is. A *tier*, in this sense, is a layer of an application that provides some specific set of functionality. Here are the three basic tiers used in database applications:

- Data—The data tier is responsible for storing your data. Typically, this will be an RDBMS such as Microsoft SQL Server, Oracle, or InterBase.
- Business—The business tier is responsible for retrieving data from the data tier in a format appropriate for the application and performing final validation of the data (also known as *enforcing business rules*). This is also the application server layer.
- Presentation—Also known as the *GUI tier*, this tier is responsible for displaying the data in an appropriate format in the client application. The presentation tier always talks to the business tier. It never talks directly to the data tier.

In traditional client/server applications, you have an architecture similar to that shown in Figure 21.1. Notice that the client libraries for data access must be located on every single client machine. This has historically been a trouble spot when deploying client/server applications due to incompatible versions of DLLs and costly configuration management. Also, because most of the business tier is located on each client, you need to update all the clients every single time you need to update a business rule.

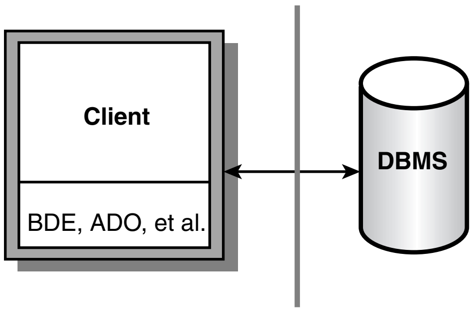


FIGURE 21.1
The traditional client/server architecture.

In multitier applications, the architecture more closely resembles that shown in Figure 21.2 Using this architecture, you'll find many benefits over the equivalent client/server application.

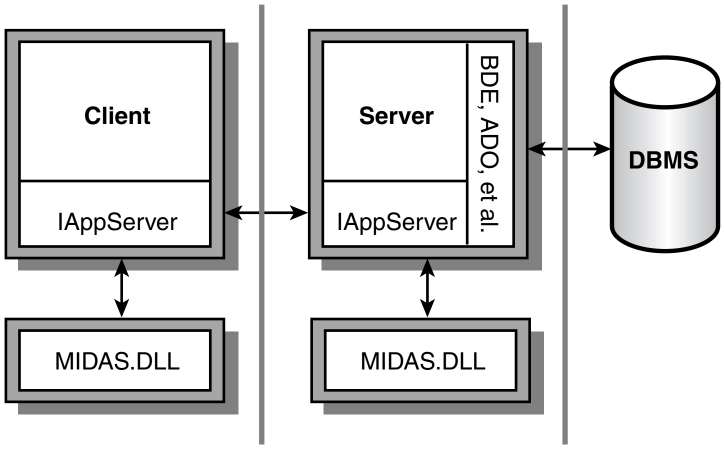


FIGURE 21.2
Multitier architecture.

Benefits of the Multitier Architecture

We list the major benefits of the multitier architecture in the next few sections.

Centralized Business Logic

In most client/server applications, each client application is required to keep track of the individual business rules for a business solution. Not only does this increase the size of the executable, but it also poses a challenge to the software developer to keep strict control over version maintenance. If user A has an older version of the application than user B, the business rules might not be performed consistently, thus resulting in logical data errors. Placing the business rules on the application server requires only one copy of the business rules to be created and maintained. Therefore, everyone using that application server will use the same copy of those business rules. In client/server applications, the RDBMS could address some of the concerns, but not all RDBMS systems provide the same set of features. Also, writing stored procedures makes your application less portable. Using a multitier approach, your business rules are hosted independent of your RDBMS, thus making database independence easier, while still providing some degree of rule enforcement for your data.

Thin-Client Architecture

In addition to the business rules mentioned, the typical client/server application also bears the burden of the majority of the data-access layer. This produces a more sizable executable, more commonly known as a *fat client*. For a Delphi database application accessing a SQL server database, you would need to install the BDE, SQL Links, and/or ODBC to access the database, along with the client libraries necessary to talk to the SQL server. After installing these files, you would then need to configure each piece appropriately. This increases the install footprint considerably. Using DataSnap, the data access is controlled by the application server, whereas the data is presented to the user by the client application. This means that you only need to distribute the client application and one DLL to help your client talk to your server. This is clearly a thin-client architecture.

Automatic Error Reconciliation

Delphi comes with a built-in mechanism to help with error reconciliation. Error reconciliation is necessary in a multitier application for the same reasons it would be necessary with cached updates. The data is copied to the client machine, where changes are made. Multiple clients can be working on the same record. Error reconciliation helps the user determine what to do with records that have changed since the user last downloaded the record. In the true Delphi spirit, if this dialog doesn't suit your needs, you can remove it and create one that does.

Briefcase Model

The briefcase model is based on the metaphor of a physical briefcase. You place your important papers in your briefcase and transport them back and forth, unpacking them when needed. Delphi provides a way to pack up all your data and take it with you on the road without requiring a live connection to the application server or the database server.

Fault Tolerance

If your server machine becomes unavailable due to unforeseen circumstances, it would be nice to dynamically change to a backup server without recompiling your client or server applications. Delphi provides functionality for this out of the box.

Load Balancing

As you deploy your client application to more people, you'll inevitably start to saturate your server's bandwidth. There are two ways to attempt to balance the network traffic: static and dynamic load balancing. For static load balancing, you would add another server machine and have one half of your clients use server A, and the other half would access server B. However, what if the clients who use server A put a greater strain on the server than those who use server

B? Using dynamic load balancing, you could address this issue by telling each client application which server to access. Many different dynamic load-balancing algorithms are available, such as random, sequential, round robin, and least network traffic. Delphi 4 and above address this by providing you with a component to implement sequential load balancing.

Typical DataSnap Architecture

Figure 21.3 shows how a typical DataSnap application looks after it's created. At the heart of this diagram is a Data Module constructed for this task. Several varieties are available. For simplicity, we'll use a COM-based one in this chapter, called the Remote Data Module (RDM). The RDM is a descendant of the classic data module available since Delphi 2. This data module is a special container that only allows non-visual components to be placed on it. The RDM is no different in this respect. In addition, the RDM is actually a COM object—or to be more precise, an *Automation object*. Services that you export from this RDM will be available for use on client machines.

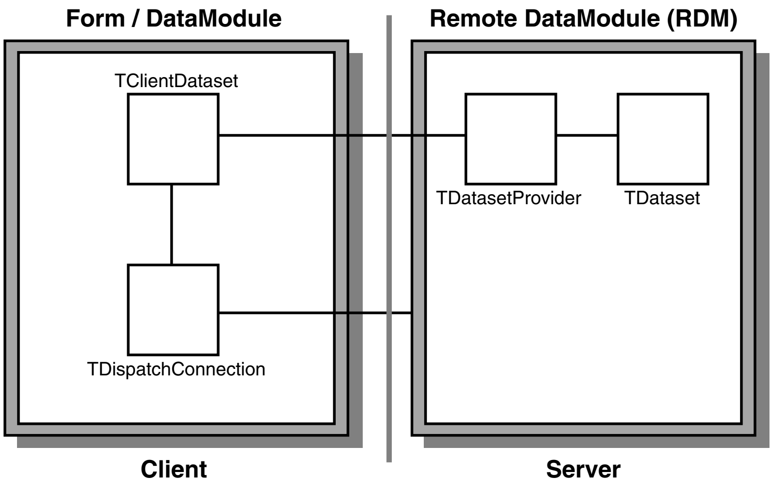
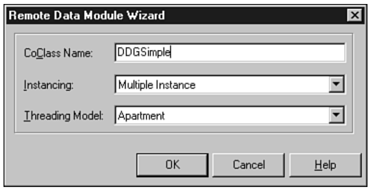


FIGURE 21.3
A typical DataSnap application.

Let's look at some of the options available to you when creating an RDM. Figure 21.4 shows the dialog box that Delphi presents when you select File, New, Remote Data Module.

Server

Now that you've seen how a typical DataSnap application is put together, we will show you how to make that happen in Delphi. We'll begin with a look at some of the choices available when setting up the server.



CoClass Name: DDGSimple
Instancing: Multiple Instance
Threading Model: Apartment

FIGURE 21.4
The New Remote Data Module dialog box.

Instancing Choices

Specifying an instancing choice affects how many copies of the server process that will be launched. Figure 21.5 shows how the choices made here control how your server behaves.

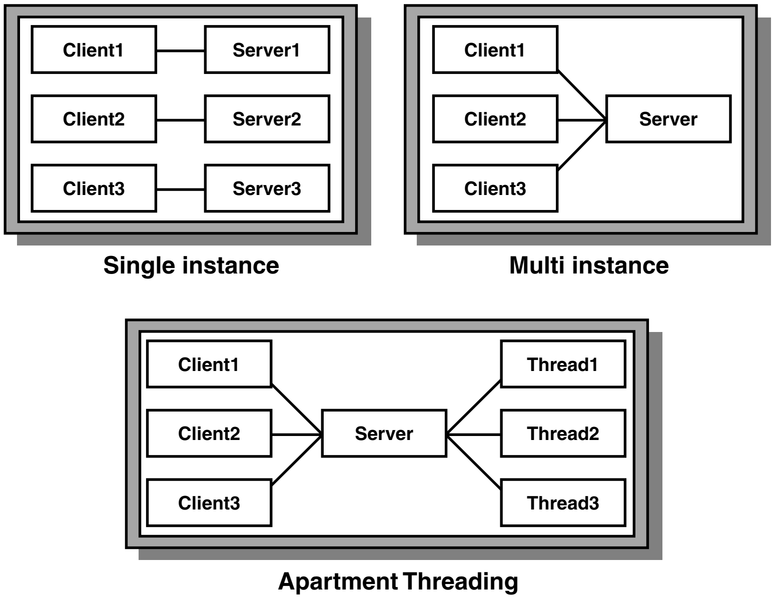


FIGURE 21.5
Server behavior based on instancing options.

Here are the different instancing choices available to a COM server:

- `ciMultiInstance`—Each client that accesses the COM server will use the same server instance. By default, this implies that one client must wait for another before being allowed to operate on the COM server. See the next section, "Threading Choices," for more detailed information on how the value specified for the Threading Model also affects this behavior. This is equivalent to serial access for the clients. All clients must share one database connection; therefore, the TDatabase.HandleShared property must be True.

- ciSingleInstance—Each client that accesses the COM server will use a separate instance. This implies that each client will consume server resources for each server instance to be loaded. This is equivalent to parallel access for the clients. If you decide to go with this choice, beware of BDE limits that could make this choice less attractive. Specifically, BDE 5.01 has a 48-process limit per machine. Because each client spawns a new server process, you can only have 48 clients connected at one time.
- ciInternal—The COM server cannot be created from external applications. This is useful when you want to control access to a COM object through a proxy layer. One example of using this instancing choice can be found in the <DELPHI>\DEMOS\MIDAS\POOLER example.

Also note that the configuration of the DCOM object has a direct effect on the object instancing mode. See the "Deploying DataSnap Applications" section for more information on this topic.

Threading Choices

The threading support in Delphi 5 saw a drastic change for the better. In Delphi 4, selecting the threading model for an EXE server was meaningless. The flag merely marked the Registry to tell COM that a DLL was capable of running under the selected threading model. With Delphi 5 and 6, the threading model choice now applies to EXE servers by allowing COM to thread the connections without using any external code. The following is a summary of the threading choices available for an RDM:

- Single—Selecting Single means that the server is only capable of handling one request at a time. When using Single, you need not worry about threading issues because the server runs in one thread and COM handles the details of synchronizing the messages for you. However, this is the worst selection you can make if you plan on having a multiuser system because client B would then need to wait for client A to finish processing before client B could even start working. This obviously isn't a good situation because client A could be doing an end-of-day summary report or some other similar time-intensive operation.
- Apartment—Selecting the Apartment threading model gives you the best of all possible worlds when combined with ciMultiInstance instancing. In this scenario, all the clients share one server process because of ciMultiInstance, but the work done on the server from one client doesn't block another client from doing work due to the Apartment threading choice. When using apartment threading, you're guaranteed that the instance data of your RDM is safe, but you need to protect access to global variables using some

thread synchronization technique, such as PostMessage(), critical sections, mutexes, semaphores, or the Delphi wrapper class TMultiReadExclusiveWriteSynchronizer. This is the preferred threading model for BDE datasets. Note that if you do use this threading model with BDE datasets, you need to place a TSession component on your RDM and set the AutoSessionName property to True to help the BDE conform to its internal requirements for threading.

- Free—This model provides even more flexibility in server processing by allowing multiple calls to be made from the client to the server simultaneously. However, along with that power comes responsibility. You must take care to protect all data from thread conflicts—both instance data and global variables. This is the preferred threading model when using ADO.
- Both—This setting is effectively the same as the Free setting, with one exception—callbacks are serialized automatically.

Data-Access Choices

Delphi 6 Enterprise comes with many different data-access choices. The BDE continues to be supported, thus allowing you to use TDBDataset components, such as TTable, TQuery, and TStoredProc. However, DBExpress provides a more flexible architecture for data access. In addition, you also have the choice of supporting ADO and having direct InterBase access through new TDataset components.

Advertising Services

The RDM is responsible for communicating which services will be available to clients. If the RDM is going to make a TQuery available for use on the client, you need to place the TQuery on the RDM along with a TDatasetProvider. The TDatasetProvider component is then tied to the TQuery via the TDatasetProvider.Dataset property. Later, when a client comes along and wants to use the data from the TQuery, it can do so by binding to the TDatasetProvider you just created. You can control which providers are visible to the client by setting the TDatasetProvider.Exported property to True or False.

If, on the other hand, you don't need an entire dataset exposed from the server and just have a need for the client to make a method call to the server, you can do that, too. While the RDM has focus, select the Edit, Add To Interface menu option and fill in the dialog box with a standard method prototype (which is simply a declaration matching a method you'll create in your implementation). You can then specify the implementation of this method in code as you always have, keeping in mind the implications of your threading model.

Client

After building the server, you need to create a client to use the services provided by the server. Let's take a look at some of the options available when building your DataSnap client.

Connection Choices

Delphi's architecture for connecting the client to the server starts with the `TDispatchConnection`. This base object is the parent of all the connection types listed later. When the connection type is irrelevant for a specific section, `TDispatchConnection` will be used to denote that fact.

`TDCOMConnection` provides core security and authentication by using the standard Windows implementation of these services. This connection type is especially useful if you're using this application in an intranet/extranet setup (that is, where the people using your application are known from the domain's perspective). You can use early binding when using DCOM, and you can use callbacks and `ConnectionPoints` easily. (You can use callbacks when using sockets, too, but you're limited to using late binding to do so.) The drawbacks of using this connection are as follows:

- Difficult configuration in many cases
- Not a firewall-friendly connection type
- Requires installation of DCOM95 for Windows 95 machines

`TSocketConnection` is the easiest connection to configure. In addition, it only uses one port for DataSnap traffic, so your firewall administrators will be happier than if they had to make DCOM work through the firewall. You must be running ScktSrvr (found in the `<DELPHI>\BIN` directory) to make this setup work, so there's one extra file to deploy and run on the server. Delphi 4 required you to have WinSock2 installed when using this connection type, which meant another installation for Windows 9*x* clients. However, if you're not using callbacks, you might want to consider setting `TSocketConnection.SupportCallbacks` to `False`. This allows you to stick with WinSock 1 on the client machines.

You can also use `TCORBAConnection` if you want to use CORBA as your transport protocol. CORBA can be thought of as the open-standard equivalent of DCOM, and it includes many features for autodiscovery, failover, and load-balancing automatically performed for your application. You'll want to look at CORBA as you migrate your DataSnap applications to allow for cross-platform and cross-language connections.

The `TWebConnection` component is also available to you. This connection component allows traffic to be transported over HTTP or HTTPS. When using this connection type, some limitations are as follows:

- Callbacks of any type aren't supported.
- The client must have `WININET.DLL` installed.
- The server machine must be running MS Internet Information Server (IIS) 4.0 or Netscape 3.6 or greater.

However, these limitations seem well worth it when you have to deliver an application across the Internet or through a firewall that's not under your control.

Delphi 6 introduced a new type of connection: the `TSOAPConnection`. This connection behaves similarly to the `WebConnection`, but connects to a DataSnap Web service. Unlike when using other DataSnap connection components, you can't use the `AppServer` property of `TSoapConnection` to call methods of the application server's interface that aren't `IAppServer` methods. Instead, to communicate with a SOAP data module on the application interface, use a separate `THTTPRIO` object.

Note that all these transports assume a valid installation of TCP/IP. The one exception to this is if you're using two Windows NT machines to communicate via DCOM. In that case, you can specify which protocol DCOM will use by running DCOMCNFG and moving the desired protocol to the top of the list on the Default Protocols tab. DCOM for Windows 9*x* only supports TCP/IP.

Connecting the Components

From the diagram in Figure 21.3, you can see how the DataSnap application communicates across tiers. This section points out the key properties and components that give the client the ability to communicate with the server.

To communicate from the client to the server, you need to use one of the `TDispatchConnection` components listed previously. Each component has properties specific only to that connection type, but all of them allow you to specify where to find the application server. The `TDispatchConnection` is analogous to the `TDatabase` component when used in client/server applications because it defines the link to the external system and serves as the conduit for other components when communicating with elements from that system.

Once you have a connection to the server, you need a way to use the services you exposed on the server. This can be accomplished by dropping a `TClientDataset` on your client and hooking it up to the `TDispatchConnection` via the `RemoteServer` property. Once this connection is made, you can view a list of the exported providers on the server by dropping down the list in the `ProviderName` property. You'll see a list of exported providers that exist on the server. In this way, the `TClientDataset` component is similar to a `TTable` in client/server applications.

You also have the ability to call custom methods that exist on the server by using the `TDispatchConnection.AppServer` property. For example, the following line of code will call the `Login` function on the server, passing two string parameters and returning a Boolean value:

```
LoginSucceeded := DCOMConnection1.AppServer.Login(UserName, Password);
```

Using DataSnap to Create an Application

Now that we've covered many of the options available when building DataSnap applications, let's use DataSnap to actually create an application to put that theory into practice.

Setting Up the Server

Let's focus on the mechanics of building the application server first. After you have created the server, we will explore how to build the client.

Remote Data Module

The Remote Data Module (RDM) is central to creating an application server. To create an RDM for a new application, select the Remote Data Module icon from the Multitier tab of the Object Repository (available by selecting File, New). A dialog box will be displayed to allow for initial customization of some options that pertain to the RDM.

The name for the RDM is important because the ProgID for this application server will be built using the project name and RDM name. For example, if the project (DPR) is named `AppServer` and the RDM name is `MyRDM`, the ProgID will be `AppServer.MyRDM`. Be sure to select the appropriate instancing and threading options based on the preceding explanations and the behavior desired for this application server.

Both `TSocketConnection` and `TWebConnection` bypass Windows' default authentication processing, so it is imperative to make sure that the only objects that run on the server are the ones that you specify. This is accomplished by marking the registry with certain values to let DataSnap know that you intended to allow these objects to run. Fortunately, all that is required to do this is to override the `UpdateRegistry` class method. See Listing 21.1 for the implementation provided by Delphi automatically when you create a new Remote DataModule.

LISTING 21.1 `UpdateRegistry` Class Method from a Remote DataModule

```
class procedure TDDGSimple.UpdateRegistry(Register:  Boolean;
 const ClassID, ProgID: string);
begin
  if Register then
  begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
  end else
  begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
```

LISTING 21.1 Continued

```
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;
```

This method gets called whenever the server gets registered or unregistered. In addition to the COM-specific registry entries that get created in the inherited `UpdateRegistry` call, you can call the `EnableXXXTransport()` and `DisableXXXTransport()` methods to mark this object as secure.

NOTE

`TSocketConnection` will only show registered, secure objects in the `ServerName` property. If you don't want to enforce security at all, uncheck the Connections, Registered Objects Only menu option in the SCKTSRVR.

Providers

The application server will be responsible for providing data to the client, so you must find a way to serve data from the server in a format that's useable on the client. Fortunately, DataSnap provides a `TDatasetProvider` component to make this step easy.

Start by dropping a `TQuery` on the RDM. If you're using a RDBMS, you'll inevitably need a `TDatabase` component set up, too. For now, you'll tie the `TQuery` to the `TDatabase` and specify a simple query in the SQL property, such as `select * from customer`. Last, drop a `TDatasetProvider` component onto the RDM and tie it to the `TQuery` via the `Dataset` property. The `Exported` property on the `DatasetProvider` determines whether this provider will be visible to clients. This property provides the ability to easily control which providers are visible at runtime, as well.

NOTE

Although the discussion in this section focuses on using the BDE-based `TDBDataset`, the same principles apply if you want to use any other `TDataset` descendant for your data access. Several possibilities exist out of the box, such as DBExpress, ADO, and InterBase Express, and several third-party components are available to access specific databases.

Registering the Server

Once the application server is built, it needs to be registered with COM to make it available for the client applications that will connect with it. The Registry entries discussed in Chapter 15, "COM Development" are also used for DataSnap servers. You just need to run the server application, and the Registry setting will be added. However, before registering the server, be sure to save the project first. This ensures that the ProgID will be correct from this point forward.

If you would rather not run the application, you can pass the parameter `/regserver` on the command line when running the application. This will just perform the registration process and immediately terminate the application. To remove the Registry entries associated with this application, you can use the `/unregserver` parameter.

Creating the Client

Now that you have a working application server, let's look at how to perform some basic tasks with the client. We will discuss how to retrieve the data, how to edit the data, how to update the database with changes made on the client, and how to handle errors during the database update process.

Retrieving Data

Throughout the course of a database application, it's necessary to bring data from the server to the client to edit that data. By bringing the data to a local cache, you can reduce network traffic and minimize transaction times. In previous versions of Delphi, you would use cached updates to perform this task. However, the same general steps still apply to DataSnap applications.

The client talks to the server via a `TDispatchConnection` component. Providing the `TDispatchConnection` the name of the computer where the application server lives accomplishes this task easily. If you use `TDCOMConnection`, you can specify the fully qualified domain name (FQDN; for example, `nt.dmiser.com`), the numeric IP address of the computer (for example, `192.168.0.2`), or the NetBIOS name of the computer (for example, `nt`). However, because of a bug in DCOM, you cannot use the name `localhost`, or even some IP addresses, reliably in all cases. If you use `TSocketConnection`, you specify numeric IP addresses in the `Address` property or the FQDN in the `Host` property. We'll take a look at the options for `TWebConnection` a little later.

Once you specify where the application server resides, you need to give the `TDispatch Connection` a way to identify that application server. This is done via the `ServerName` property. Assigning the `ServerName` property fills in the `ServerGUID` property for you. The `ServerGUID` property is the most important part. As a matter of fact, if you want to deploy your client application in the most generic manner possible, be sure to delete the `ServerName` property and just use the `ServerGUID`.

NOTE

If you use TDCOMConnection, the ServerName list will only display the list of servers that are registered on the current machine. However, TSocketConnection is smart enough to display the list of application servers registered on the remote machine.

At this point, setting TDispatchConnection.Connected to True will connect you to the application server.

Now that you have the client talking to the server, you need a way to use the provider you created on the server. Do this by using the TClientDataset component. A TClientDataSet is used to link to a provider (and, thus, the TQuery that is linked to the provider) on the server.

First, you must tie the TClientDataSet to the TDispatchConnection by assigning the RemoteServer property of the TClientDataSet. Once you've done that, you can get a list of the available providers on that server by looking at the list in the ProviderName property.

At this point, everything is now set up properly to open a ClientDataset.

Because the TClientDataSet is a virtual TDataset descendant, you can build on many of the techniques that you've already learned using the TDBDataset components in client/server applications. For example, setting Active to True opens the TClientDataSet and displays the data. The difference between this and setting TTable.Active to True is that the TClientDataSet is actually getting its data from the application server.

Editing Data on the Client

All the records passed from the server to the TClientDataSet are stored in the Data property of the TClientDataSet. This property is a variant representation of the DataSnap data packet. The TClientDataset knows how to decode this data packet into a more useful format. The reason the property is defined as a variant is because of the limited types available to the COM subsystem when using type library marshaling.

As you manipulate the records in the TClientDataset, a copy of the inserted, modified, or deleted records gets placed in the Delta property. This allows DataSnap to be extremely efficient when it comes to applying updates back to the application server, and eventually the database. Only the changed records need to be sent back to the application server.

The format of the Delta property is also very efficient. It stores one record for every insert or delete, and it stores two records for every update. The updated records are stored in an efficient manner, as well. The unmodified record is provided in the first record, whereas the corresponding modified record is stored next. However, only the changed fields are stored in the modified record to save on storage.

One interesting aspect of the `Delta` property is that it's compatible with the `Data` property. In other words, it can be assigned directly to another `ClientDataset` component's `Data` property. This will allow you to investigate the current contents of the `Delta` property at any given time.

Several methods are available to deal with the editing of data on the `TClientDataset`. We'll refer to these methods as change control methods. The *change control* methods allow you to modify the changes made to the `TClientDataset` in a variety of ways.

NOTE

`TClientDataset` has proven useful in more ways than originally intended. It also serves as an excellent method for storing in-memory tables, which has nothing to do with DataSnap specifically. Additionally, because of the way it exposes data through the Data and Delta properties, it has proven useful in a variety of OOP pattern implementations. It is beyond the scope of the chapter to discuss these techniques. However, you will find white papers on these topics at `http://www.xapware.com` or `http://www.xapware.com/ddg`.

Undoing Changes

Most users have used a word-processing application that permits the Undo operation. This operation takes your most previous action and rolls it back to the state right before you started. Using `TClientDataset`, you can call `cdsCustomer.UndoLastChange()` to simulate that behavior. The undo stack is unlimited, allowing the user to continue to back up all the way to the beginning of the editing session if so desired. The parameter you pass to this method specifies whether the cursor is positioned to the record being affected.

If the user wanted to get rid of all her updates at once, there's an easier way than calling `UndoLastChange()` repeatedly. You can simply call `cdsCustomer.CancelUpdates()` to cancel all changes that have been made in a single editing session.

Reverting to the Original Version

Another possibility is to allow the user to restore a specific record back to the state it was in when the record was first retrieved. Do this by calling `cdsCustomer.RevertRecord()` while the `TClientDataset` is positioned on the record you intend to restore.

Client-Side Transactions: `SavePoint`

The `ClientDataset.SavePoint` property provides the ability to use client-side transactions. This property is ideal for developing what-if scenarios for the user. The act of retrieving the value of the `SavePoint` property stores a snapshot of the data at that point in time. The user can continue to edit as long as needed. If, at some point, the user decides that the baseline set of data is actually what she wanted, that saved variable can be assigned back to SavePoint and the TClientDataset is returned back to the same state it was in at the time when the initial snapshot was taken. It's worth noting that you can have multiple, nested levels of SavePoint for a complex scenario as well.

CAUTION

A word of caution about SavePoint is in order: You can invalidate a SavePoint by calling UndoLastChange() past the point that's currently saved. For example, assume that the user edits two records and issues a SavePoint. At this point, the user edits another record. However, she uses UndoLastChange() to revert changes twice in a row. Because the TClientDataset state is now in a state prior to the SavePoint, the SavePoint is in an undefined state.

Reconciling Data

After you've finished making changes to the local copy of data in the TClientDataset, you'll need to signal your intent to apply these changes back to the database. This is done by calling cdsCustomer.ApplyUpdates(). At this point, DataSnap will take the Delta from cdsCustomer and pass it to the application server, where DataSnap will apply these changes to the database server using the reconciliation mechanism that you chose for this dataset. All updates are performed inside the context of a transaction. We'll cover how errors are handled during this process shortly.

The parameter you pass into ApplyUpdates() specifies the number of errors the update process will allow before considering the update to be bad, and subsequently, roll back all the changes that have been made. The word *errors* here refers to key violation errors, referential integrity errors, or any other business logic or database errors. If you specify zero for this parameter, you're telling DataSnap that you won't tolerate any errors. Therefore, if an error does occur, all the changes you made will not be committed to the database. This is the setting that you'll use most often because it most closely matches solid database guidelines and principles.

However, if you want, you can specify that a certain number of errors can occur, while still committing all the records that were successful. The ultimate extension of this concept is to pass -1 as the parameter to ApplyUpdates(). This tells DataSnap that it should commit every single record that it can, regardless of the number of errors encountered along the way. In other words, the transaction will always commit when using this parameter.

If you want to take ultimate control over the update process—including changing the SQL that will execute for an insert, update, or delete—you can do so in the TDatasetProvider.Before UpdateRecord() event. For example, when a user wants to delete a record, you might not want to actually perform a delete operation on the database. Instead, a flag is set to tell applications that this record isn't available. Later, an administrator can review these deletions and commit the physical delete operation. The following example shows how to do this:

```
procedure TDataModule1.Provider1BeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataset; DeltaDS: TClientDataset; UpdateKind: TUpdateKind;
  var Applied: Boolean);
begin
  if UpdateKind=ukDelete then
  begin
    Query1.SQL.Text:='update CUSTOMER set STATUS="DEL" where ID=:ID';
    Query1.Params[0].Value:=DeltaDS.FieldByName('ID').OldValue;
    Query1.ExecSQL;
    Applied:=true;
  end;
end;
```

You can create as many queries as you want, controlling the flow and content of the update process based on different factors, such as `UpdateKind` and values in the `Dataset`. When inspecting or modifying records of the `DeltaDS`, be sure to use the `OldValue` and `NewValue` properties of the appropriate `TField`. Using `the TField.As`*`XXX`* properties will yield unpredictable results.

In addition, you can enforce business rules here or avoid posting a record to the database altogether. Any exception you raise here will wind its way through DataSnap's error-handling mechanism, which we'll cover next.

After the transaction is finished, you get an opportunity to deal with errors. The error stops at events on both the server and the client, giving you a chance to take corrective action, log the error, or do anything else you want to with it.

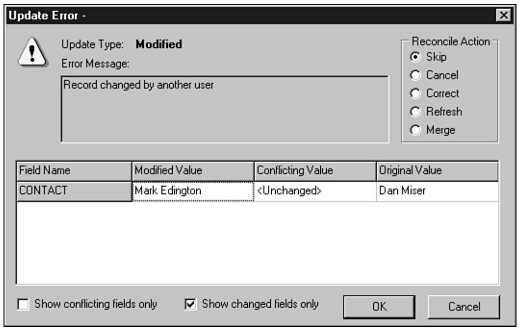
The first stop for the error is the `DatasetProvider.OnUpdateError` event. This is a great place to deal with errors that you're expecting or can resolve without further intervention from the client.

The final destination for the error is back on the client, where you can deal with the error by letting the user help determine what to do with the record. You do this by assigning an event handler to the `TClientDataset.OnReconcileError` event.

This is especially useful because DataSnap is based on an optimistic record-locking strategy. This strategy allows multiple users to work on the same record at the same time. In general, this causes conflicts when DataSnap tries to reconcile the data back to the database because the record has been modified since it was retrieved.

Using Borland's Error Reconciliation Dialog Box

Fortunately, Borland provides a standard error reconciliation dialog box that you can use to display the error to the user. Figure 21.6 shows this dialog box. The source code is also provided for this unit, so you can modify it if it doesn't suit your needs. To use this dialog box, select File, New in Delphi's main menu and then select Reconcile Error Dialog from the Dialogs page. Remember to remove this unit from the Autocreate Forms list; otherwise, you'll receive compile errors.


Update Error -

Update Type: **Modified**
Error Message:
Record changed by another user

Reconcile Action
[✓ Skip] Cancel Correct Refresh Merge

Field Name	Modified Value	Conflicting Value	Original Value
CONTACT	Mark Edington	<Unchanged>	Dan Miser

☐ Show conflicting fields only
☑ Show changed fields only

FIGURE 21.6
Reconcile Error dialog box in action.

The main functionality of this unit is wrapped up in the function `HandleReconcileError()`. A high degree of correlation exists between the `OnReconcileError` event and the `HandleReconcileError` function. As a matter of fact, the typical course of action in the `OnReconcileError` event is to call the `HandleReconcileError` function. By doing this, the application allows the end user on the client machine to interact with the error reconciliation process on the server machine and specify how these errors should be handled. Here's the code:

```
procedure TMyForm.CDSReconcileError(Dataset: TCustomClientDataset;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
begin
  Action:=HandleReconcileError(Dataset, UpdateKind, E);
end;
```

The value of the `Action` parameter determines what DataSnap will do with this record. We'll touch on some other factors that affect which actions are valid at this point a little later. The following list shows the valid actions:

- `raSkip`—Do not update this specific database record. Leave the changed record in the client cache.

- `raMerge`—Merge the fields from this record into the database record. This record won't apply to records that were inserted.
- `raCorrect`—Update the database record with the values you specify. When selecting this action in the Reconcile Error dialog box, you can edit the values in the grid. You cannot use this method if another user changed the database record.
- `raCancel`—Don't update the database record. Remove the record from the client cache.
- `raRefresh`—Update the record in the client cache with the current record in the database.
- `raAbort`—Abort the entire update operation.

Not all these options make sense (and therefore won't be displayed) in all cases. One requirement to have the `raMerge` and `raRefresh` actions available is that DataSnap can identify the record via the primary key of the database. This is done by setting the `TField.ProviderFlags.pfInKey` property to `True` on the `TDataset` component of the RDM for all fields in your primary key.

More Options to Make Your Application Robust

Once you master these basics, the inevitable question is "What next?" This section is provided to give you some more insight into DataSnap and how you can use these features to make your applications act as you want them to act.

Client Optimization Techniques

The model of retrieving data is fairly elegant. However, because the `TClientDataset` stores all its records in memory, you need to be very careful about the resultsets you return to the `TClientDataSet`. The cleanest approach is to ensure that the application server is well designed and only returns the records the user is interested in. Because the real world seldom follows the utopian solution, you can use the following technique to help throttle the number of records you retrieve at one time to the client.

Limiting the Data Packet

When opening a `TClientDataSet`, the server retrieves the number of records specified in the `TClientDataSet.PacketRecords` property at one time. However, DataSnap will retrieve enough records to fill all available visual controls with data. For example, if you have a `TDBGrid` on a form that can display 10 records at once, and you specify a value of `5` for `PacketRecords`, the initial fetch of data will contain 10 records. After that, the data packet will contain just 5 records per fetch. If you specify `-1` for this property (the default), all records will be transferred. If you specify a value greater than zero for `PacketRecords`, this introduces state to your application. This is because of the requirement that the app server must keep track of each client's cursor position so the app server can return the appropriate packet of records to the client requesting a packet. However, you can keep track of the state on the client, passing the last record position to the server, as appropriate. For a simple example, look at this code, which does exactly that:

```
Server RDM:
procedure TStateless.DataSetProvider1BeforeGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
  begin
    DataSet.Open;
    if VarIsEmpty(OwnerData) then
      DataSet.First
    else
    begin
      while not DataSet.Eof do
      begin
        if DataSet.FieldByName('au_id').Value = OwnerData then
          break;
      end;
    end;
  end;
end;

procedure TStateless.DataSetProvider1AfterGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
  begin
    OwnerData := Dataset.FieldValues['au_id'];
    DataSet.Close;
  end;
end;

Client:
procedure TForm1.ClientDataSet1BeforeGetRecords(Sender: TObject;
  var OwnerData:  OleVariant);
begin
  // KeyValue is a private OleVariant variable
  if not (Sender as TClientDataSet).Active then
    KeyValue := Unassigned;
  OwnerData := KeyValue;
end;

procedure TForm1.ClientDataSet1AfterGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
```

```
begin
  KeyValue := OwnerData;
end;
```

One last point when using partial fetching is that executing `TClientDataSet.Last()` retrieves the rest of the records left in the resultset. This can be done innocently by pressing Ctrl+End in the `TDBGrid`. To work around this problem, you should set `TClientDataSet.FetchOnDemand` to `False`. This property controls whether a data packet will be retrieved automatically when the user has read through all the existing records on the client. To emulate that behavior in code, you can use the `GetNextPacket()` method, which will return the next data packet for you.

NOTE

Note that the previous code sample walks through the dataset until it finds the proper record. This is done so that unidirectional datasets such as DBExpress can use this same code without modification. Of course, there are many ways to find the proper record, such as modifying an SQL statement or parameterizing a query, but this sample concentrates on the mechanics of passing around the key between client and server.

Using the Briefcase Model

Another optimization to reduce network traffic is to use the briefcase model support offered with DataSnap. Do this by assigning a filename to the `TClientDataset.Filename` property. If the file specified in this property exists, the `TClientDataSet` will open up the local copy of the file as opposed to reading the data directly from the application server. In addition to allowing users to work with files while disconnected from the network, this is tremendously useful for items that rarely change, such as lookup tables.

TIP

If you specify a `TClientDataset.Filename` that has an `.XML` extension, the data packet will be stored in XML format, enabling you to use any number of XML tools available to work on the briefcase file.

Sending Dynamic SQL to the Server

Some applications require modification to the underlying `TDataset`'s core properties, such as the `SQL` property of the `TQuery`, from the client. As long as solid multitier principles are followed, this can actually be a very efficient and elegant solution. Delphi makes this task trivial to accomplish.

Two steps are required to allow for ad hoc queries. First, you simply assign the query statement to the `TClientDataset.CommandText` property. Second, you must also include the `poAllowCommandText` option in the `DatasetProvider.Options` property. When you open the `TClientDataSet` or call `TClientDataSet.Execute()`, the `CommandText` is passed across to the server. This same technique also works if you want to change the table or stored procedure name on the server.

Application Server Techniques

DataSnap now has many different events for you to customize the behavior of your application. `Before`*`XXX`* and `After`*`XXX`* events exist for just about every method on the IAppServer interface. These two events in particular will be useful as you migrate your application server to be completely stateless.

Resolving Record Contention

The preceding discussion of the resolving mechanism included a brief mention that two users working on the same record would cause an error when the second user tried to apply the record back to the database. Fortunately, you have full control over detecting this collision.

The `TDatasetProvider.UpdateMode` property is used to generate the SQL statement that will be used to check whether the record has changed since it was last retrieved. Consider the scenario in which two users edit the same record. Here's how `DatasetProvider.UpdateMode` affects what happens to the record for each user:

- `upWhereAll`—This setting is the most restrictive setting but provides the greatest deal of assurance that the record is the same one the user retrieved initially. If two users edit the same record, the first user will be able to update the record, whereas the second user will receive the infamous `Another user changed the record.` error message. If you want to further refine which fields are used to perform this check, you can remove the `pfInWhere` element from the corresponding `TField.ProviderFlags` property.
- `upWhereChanged`—This setting allows the two users to actually edit the same record at the same time; as long as both users edit different fields in the same record, there will be no collision detection. For example, if user A modifies the `Address` field and updates the record, user B can still modify the `BirthDate` field and update the record successfully.
- `upWhereKeyOnly`—This setting is the most forgiving of all. As long as the record exists on the database, every user's change will be accepted. This will always overwrite the existing record in the database, so it can be viewed as a way to provide "last one in wins" functionality.

Miscellaneous Server Options

Quite a few more options are available in the TDatasetProvider.Options property to control how the DataSnap data packet behaves. For example, adding poReadOnly will make the dataset read-only on the client. Specifying poDisableInserts, poDisableDeletes, or poDisableEdits prevents the client from performing that operation and triggers the corresponding OnEditError or OnDeleteError event to be fired on the client.

When using nested datasets, you can have updates or deletes cascade from the master record to the detail records if you add poCascadeUpdates or poCascadeDeletes to the DatasetProvider.Options property. Using this property requires your back-end database to support cascading referential integrity.

One shortcoming in previous versions of DataSnap was the inability to easily merge changes made on the server into your TClientDataset on the client. The user had to resort to using RefreshRecord (or possibly Refresh to repopulate the entire dataset in some cases) to achieve this.

By setting DatasetProvider.Options to include poPropogateChanges, all the changes made to your data on the application server (for example, in the DatasetProvider.BeforeUpdateRecord event to enforce a business rule) are now automatically brought back into the TClientDataSet. Furthermore, setting TDatasetProvider.Options to include poAutoRefresh will automatically merge AutoIncrement and default values back into the TClientDataSet.

CAUTION

The poAutoRefresh option is non-functional in Delphi 5 and 6. poAutoRefresh will only work with a later version of Delphi that includes the fix for this bug. The workaround in the meantime is to either call Refresh() for your TClientDatasets or take control of the entire process of applying updates yourself.

The entire discussion of the reconciliation process thus far has revolved around the default SQL-based reconciliation. This means that all the events on the underlying TDataset will not be used during the reconciliation process. The TDatasetProvider.ResolveToDataset property was created to use these events during reconciliation. For example, if TDatasetProvider.ResolveToDataset is true, most of the events on the TDataset will be triggered. Be aware that the events used are only called when applying updates back to the server. In other words, if you have a TQuery.BeforeInsert event defined on the server, it will only fire on the server once you call TClientDataSet.ApplyUpdates. The events don't integrate into the corresponding events of the TClientDataSet.

Dealing with Master/Detail Relationships

No discussion of database applications would be complete without at least a mention of master/detail relationships. With DataSnap, you have two choices for dealing with master/detail.

Nested Datasets

One option for master/detail relationships is nested datasets. Nested datasets allow a master table to actually contain detail datasets. In addition to updating master and detail records in one transaction, they allow for storage of all master and detail records to be stored in one briefcase file, and you can use the enhancements to `DBGrid` to pop up detail datasets in their own windows. A word of caution if you do decide to use nested datasets: All the detail records will be retrieved and brought over to the client when selecting a master record. This will become a possible performance bottleneck if you nest several levels of detail datasets. For example, if you retrieve just one master record that has 10 detail records, and each detail record has three detail records linked to the first level detail, you would retrieve 41 records initially. When using client-side linking, you would only retrieve 14 records initially and obtain the other grandchild records as you scrolled through the detail `TClientDataSet`.

In order to set up a nested dataset relationship, you need to define the master/detail relationship on the application server. This is done using the same technique you've been using in client/server applications—namely, defining the SQL statement for the detail `TQuery`, including the link parameter. Here's an example:

```
"select * orders where custno=:custno"
```

You then assign the `TQuery.Datasource` for the detail `TQuery` to point to a `TDatasource` component that's tied to the master `TDataset`. Once this relationship is set up, you only need to export the `TDatasetProvider` that's tied to the master dataset. DataSnap is smart enough to understand that the master dataset has detail datasets linked to it and will therefore send the detail datasets across to the client as a `TDatasetField`.

On the client, you assign the master `TClientDataset.ProviderName` property to the master provider. Then, you add persistent fields to the `TClientDataset`. Notice the last field in the Fields Editor. It contains a field named the same as the detail dataset on the server and is declared as a `TDatasetField` type. At this point, you have enough information to use the nested dataset in code. However, to make things really easy, you can add a detail `TClientDataset` and assign its `DatasetField` property to the appropriate `TDatasetField` from the master. It's important to note here that you didn't set any other properties on the detail `TClientDataset`, such as `RemoteServer`, `ProviderName`, `MasterSource`, `MasterFields`, or `PacketRecords`. The only property you set was the `DatasetField` property. At this point, you can bind data-aware controls to the detail `TClientDataset` as well.

After you've finished working with the data in the nested dataset, you need to apply the updates back to the database. This is done by calling the master `TClientDataset`'s `ApplyUpdates()` method. DataSnap will apply all the changes in the master `TClientDataset`, which includes the detail datasets, back to the server inside the context of one transaction.

You'll find an example on the book's CD-ROM in the directory for this chapter under `\NestCDS`.

Client-Side Linking

Recall that some cautions were mentioned earlier regarding using nested datasets. The alternative to using nested datasets is to create the master/detail relationship on the client side. In order to create a master/detail link using this method, you simply create a `TDataset` and `TDatasetProvider` for the master and the detail on the server.

On the client, you bind two `TClientDataset` components to the datasets that you exported on the server. Then, you create the master/detail relationship by assigning the detail `TClientDataset.MasterSource` property to the `TDatasource` component that points to the master `TClientDataset`.

Setting `MasterSource` on a `TClientDataset` sets the `PacketRecords` property to zero. When `PacketRecords` equals zero, it means that DataSnap should just return the metadata information for this `TClientDataset`. However, when `PacketRecords` equals zero in the context of a master/detail relationship, the meaning changes. DataSnap will now retrieve the records for the detail dataset for each master record. In summary, leave the `PacketRecords` property set to the default value.

In order to reconcile the master/detail data back to the database in one transaction, you need to write your own `ApplyUpdates` logic. This isn't as simple as most tasks in Delphi, but it does give you full flexible control over the update process.

Applying updates to a single table is usually triggered by a call to `TClientDataset.ApplyUpdates`. This method sends the changed records from the `ClientDataset` to its provider on the middle tier, where the provider will then write the changes to the database. All this is done within the scope of a transaction and is accomplished without any intervention from the programmer. To do the same thing for master/detail tables, you must understand what Delphi is doing for you when you make that call to `TClientDataset.ApplyUpdates`.

Any changes you make to a `TClientDataset` are stored in the `Delta` property. The `Delta` property contains all the information that will eventually be written to the database. The following code illustrates the update process for applying `Delta` properties back to the database. Listings 21.2 and 21.3 show the relevant sections of the client and server for applying updates to a master/detail setup.

LISTING 21.2 Client Updates to Master/Detail

```
procedure TClientDM.ApplyUpdates;
var
  MasterVar, DetailVar: OleVariant;
begin
  Master.CheckBrowseMode;
  Detail_Proj.CheckBrowseMode;
  if Master.ChangeCount > 0 then
    MasterVar := Master.Delta else
    MasterVar := NULL;
  if Detail.ChangeCount > 0 then
    DetailVar := Detail.Delta else
    DetailVar := NULL;
  RemoteServer.AppServer.ApplyUpdates(DetailVar, MasterVar);
  { Reconcile the error datapackets. Since we allow 0 errors, only one error
    packet can contain errors. If neither packet contains errors then we
    refresh the data.}
  if not VarIsNull(DetailVar) then
    Detail.Reconcile(DetailVar) else
  if not VarIsNull(MasterVar) then
    Master.Reconcile(MasterVar) else
  begin
    Detail.Reconcile(DetailVar);
    Master.Reconcile(MasterVar);
    Detail.Refresh;
    Master.Refresh;
  end;
end;
```

LISTING 21.3 Server Updates to Master/Detail

```
procedure TServerRDM.ApplyUpdates(var DetailVar, MasterVar: OleVariant);
var
  ErrCount: Integer;
begin
Database.StartTransaction;
  try
    if not VarIsNull(MasterVar) then
    begin
      MasterVar := cdsMaster.Provider.ApplyUpdates(MasterVar, 0, ErrCount);
      if ErrCount > 0 then
        SysUtils.Abort;    // This will cause Rollback
    end;
    if not VarIsNull(DetailVar) then
```

LISTING 21.3 Continued

```
    begin
      DetailVar := cdsDetail.Provider.ApplyUpdates(DetailVar, 0, ErrCount);
      if ErrCount > 0 then
        SysUtils.Abort;    // This will cause Rollback
    end;
    Database.Commit;
  except
    Database.Rollback
  end;
end;
```

Although this method works quite well, it really doesn't provide for opportunities for code reuse. This would be a good time to extend Delphi and provide easy reuse. Here are the main steps required to abstract the update process:

1. Place the deltas for each CDS in a variant array.
2. Place the providers for each CDS in a variant array.
3. Apply all the deltas in one transaction.
4. Reconcile the error datapackets returned in the previous step and refresh the data.

The result of this abstraction is provided in the utility unit shown in Listing 21.4.

LISTING 21.4 A Unit Providing Utility Routines and Abstraction

```
unit CDSUtil;

interface

uses
  DbClient, DbTables;

function RetrieveDeltas(const cdsArray : array of TClientDataset): Variant;
function RetrieveProviders(const cdsArray : array of TClientDataset): Variant;
procedure ReconcileDeltas(const cdsArray : array of TClientDataset;
                          vDeltaArray: OleVariant);

procedure CDSApplyUpdates(ADatabase : TDatabase; var vDeltaArray: OleVariant;
                          const vProviderArray: OleVariant);

implementation

uses
  SysUtils, Provider, Midas, Variants;
```

LISTING 21.4 Continued

```
type
  PArrayData = ^TArrayData;
  TArrayData = array[0..1000] of Olevariant;

{Delta is the CDS.Delta on input. On return, Delta will contain a data packet}
{containing all of the records that could not be applied to the database.}
{Remember Delphi needs the provider name, so it is passed in the first}
{element of the AProvider variant.}
procedure ApplyDelta(AProvider: OleVariant; var Delta : OleVariant);
var
  ErrCount : integer;
  OwnerData:  OleVariant;
begin
  if not VarIsNull(Delta) then
  begin
    // ScktSrvr does not support early-binding
    Delta := (IDispatch(AProvider[0]) as IAppServer).AS_ApplyUpdates(
                AProvider[1], Delta, 0, ErrCount, OwnerData);
    if ErrCount > 0 then
      SysUtils.Abort;  // This will cause Rollback in the calling procedure
  end;
end;

{Server call}
procedure CDSApplyUpdates(ADatabase : TDatabase; var vDeltaArray: OleVariant;
  const vProviderArray: OleVariant);
var
  i : integer;
  LowArr, HighArr: integer;
  P: PArrayData;
begin
  {Wrap the updates in a transaction. If any step results in an error, raise}
  {an exception, which will Rollback the transaction.}
  ADatabase.Connected:=true;
  ADatabase.StartTransaction;
  try
    LowArr:=VarArrayLowBound(vDeltaArray,1);
    HighArr:=VarArrayHighBound(vDeltaArray,1);
    P:=VarArrayLock(vDeltaArray);
    try
      for i:=LowArr to HighArr do
        ApplyDelta(vProviderArray[i], P^[i]);
finally
      VarArrayUnlock(vDeltaArray);
    end;
```

LISTING 21.4 Continued

```
    ADatabase.Commit;
  except
    ADatabase.Rollback;
  end;
end;

{Client side calls}
function RetrieveDeltas(const cdsArray : array of TClientDataset): Variant;
var
  i : integer;
  LowCDS, HighCDS : integer;
begin
  Result:=NULL;
  LowCDS:=Low(cdsArray);
  HighCDS:=High(cdsArray);
  for i:=LowCDS to HighCDS do
    cdsArray[i].CheckBrowseMode;

  Result:=VarArrayCreate([LowCDS, HighCDS], varVariant);
  {Setup the variant with the changes (or NULL if there are none)}
  for i:=LowCDS to HighCDS do
  begin
    if cdsArray[i].ChangeCount>0 then
      Result[i]:=cdsArray[i].Delta else
      Result[i]:=NULL;
  end;
end;

{If we're using Delphi 5 or greater, then we need to return the provider name
 AND the AppServer from this function. We will use ProviderName to call
 AS_ApplyUpdates in the CDSApplyUpdates function later.}
function RetrieveProviders(const cdsArray : array of TClientDataset): Variant;
var
  i: integer;
  LowCDS, HighCDS: integer;
begin
  Result:=NULL;
  LowCDS:=Low(cdsArray);
  HighCDS:=High(cdsArray);

  Result:=VarArrayCreate([LowCDS, HighCDS], varVariant);
  for i:=LowCDS to HighCDS do
    Result[i]:=VarArrayOf([cdsArray[i].AppServer, cdsArray[i].ProviderName]);
end;
```

Listing 21.4 Continued

```
procedure ReconcileDeltas(const cdsArray : array of TClientDataset;
  vDeltaArray: OleVariant);
var
  bReconcile : boolean;
  i: integer;
  LowCDS, HighCDS : integer;
begin
  LowCDS:=Low(cdsArray);
  HighCDS:=High(cdsArray);

  {If the previous step resulted in errors, Reconcile the error datapackets.}
  bReconcile:=false;
  for i:=LowCDS to HighCDS do
    if not VarIsNull(vDeltaArray[i]) then begin
      cdsArray[i].Reconcile(vDeltaArray[i]);
      bReconcile:=true;
      break;
    end;

  {Refresh the Datasets if needed}
  if not bReconcile then
    for i:=HighCDS downto LowCDS do begin
      cdsArray[i].Reconcile(vDeltaArray[i]);
      cdsArray[i].Refresh;
    end;
end;

end.
```

Listing 21.5 shows a reworking of the previous example using the `CDSUtil` unit.

Listing 21.5 A Rework of the Previous Example Using `CDSUtil.pas`

```
procedure TForm1.btnApplyClick(Sender: TObject);
var
  vDelta: OleVariant;
  vProvider: OleVariant;
  arrCDS: array[0..1] of TClientDataset;
begin
  arrCDS[0]:=cdsMaster;  // Set up ClientDataset array
  arrCDS[1]:=cdsDetail;

  vDelta:=RetrieveDeltas(arrCDS);                 // Step 1
  vProvider:=RetrieveProviders(arrCDS);           // Step 2
```

Listing 21.5 Continued

```
  DCOMConnection1.ApplyUpdates(vDelta, vProvider); // Step 3
  ReconcileDeltas(arrCDS, vDelta);                // Step 4
end;

procedure TServerRDM.ApplyUpdates(var vDelta, vProvider: OleVariant);
begin
  CDSApplyUpdates(Database1, vDelta, vProvider);  // Step 3
end;
```

You can use this unit in either two-tier or three-tier applications. To move from a two-tier to a three-tier approach, you would export a function on the server that calls `CDSApplyUpdates` instead of calling `CDSApplyUpdates` on the client. Everything else on the client remains the same.

You'll find an example on the book's CD-ROM in the directory for this chapter under `\MDCDS`.

Real-World Examples

Now that we have the basics out of the way, let's look at how DataSnap can help you by exploring several real-world examples.

Joins

Writing a relational database application depends heavily on walking the relationships between tables. Often, you'll find it convenient to represent your highly normalized data in a view that's more flattened than the underlying data structure. However, updating the data from these joins takes some extra care on your end.

One-Table Update

Applying updates to a joined query is a special case in database programming, and DataSnap is no exception. The problem lies in the join query itself. Although some join queries will produce data that could be automatically updated, others will never conform to rules that will allow automatic retrieval, editing, and updating of the underlying data. To that end, Delphi currently forces you to resolve updates to join queries yourself.

For joins that require only one table to be updated, Delphi can handle most of the updating details for you. Here are the steps required in order to write one table back to the database:

1. Add persistent fields to the joined `TQuery`.
2. Set TQuery.`TField.ProviderFlags=[]` for every field of the table that you won't be updating.

3. Write the following code in the `DatasetProvider.OnGetTableName` event to tell DataSnap which table you want to update. Keep in mind that this event makes it easier to specify the table name, although you could do the same thing in previous versions of Delphi by using the `DatasetProvider.OnGetDatasetProperties` event:

```
procedure TJoin1Server.prvJoinGetTableName(Sender: TObject;
  DataSet: TDataSet; var TableName: String);
begin
  TableName := 'Emp';
end;
```

By doing this, you're telling the `ClientDataset` to keep track of the table name for you. Now when you call `ClientDataset1.ApplyUpdates()`, DataSnap knows to try and resolve to the table name that you specified, as opposed to letting DataSnap try and figure out what the table name might be.

An alternative approach would be to use a `TUpdateSQL` component that only updates the table of interest. This allows the `TQuery.UpdateObject` to be used during the reconciliation process and more closely matches the process used in traditional client/server applications.

NOTE

Not all `TDatasets` have an `UpdateObject` property. However, you can still use the same approach because of the rework done to `TUpdateSQL`. Simply define your SQL for each action (delete, insert, modify) and use code similar to the following:

```
procedure TForm1.DataSetProvider1BeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  UpdateSQL1.DataSet := DeltaDS;
  UpdateSQL1.SetParams(UpdateKind);
  ADOCommand1.CommandText := UpdateSQL1.SQL[UpdateKind].Text;
  ADOCommand1.Parameters.Assign(UpdateSQL1.Query[UpdateKind].Params);
  ADOCommand1.Execute;
  Applied := true;
end;
```

You'll find an example on the book's CD-ROM in the directory for this chapter under `\Join1`.

Multitable Update

For more complex scenarios, such as allowing the editing and updating of multiple tables, you need to write some code yourself. There are two approaches to solving this problem:

- The older method of using `DatasetProvider.BeforeUpdateRecord()` to break the data packet apart and apply the updates to the underlying tables
- The newer method of applying updates by using the `UpdateObject` property

When using cached updates with a multitable join, you need to configure one `TUpdateSQL` component for each table that will be updated. Because the `UpdateObject` property can only be assigned to one `TUpdateSQL` component, you needed to link all the `TUpdateSQL.Dataset` properties to the joined dataset programmatically in `TQuery.OnUpdateRecord` and call `TUpdateSQL.Apply` to bind the parameters and execute the underlying SQL statement. In this case, the dataset you're interested in is the `Delta` dataset. This dataset is passed as a parameter into the `TQuery.OnUpdateRecord` event.

All you need to do is assign the `SessionName` and `DatabaseName` properties to allow the update to occur in the same context as other transactions and tie the Dataset property to the Delta that is passed to the event. The resulting code for the `TQuery.OnUpdateRecord` event is shown in Listing 21.6.

LISTING 21.6 Join Using a `TUpdateSQL`

```
procedure TJoin2Server.JoinQueryUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  usqlEmp.SessionName := JoinQuery.SessionName;
  usqlEmp.DatabaseName := JoinQuery.DatabaseName;
  usqlEmp.Dataset := Dataset;
  usqlEmp.Apply(UpdateKind);

  usqlFTEmp.SessionName := JoinQuery.SessionName;
  usqlFTEmp.DatabaseName := JoinQuery.DatabaseName;
  usqlFTEmp.Dataset := Dataset;
  usqlFTEmp.Apply(UpdateKind);

  UpdateAction := uaApplied;
end;
```

Because you've complied with the rules of updating data within the DataSnap architecture, the whole update process is seamlessly triggered as it always is in DataSnap, with a call to `ClientDataset1.ApplyUpdates(0);`.

You'll find an example on the book's CD-ROM in the directory for this chapter under `\Join2`.

DataSnap on the Web

Even with the introduction of Kylix, Delphi is tied to the Windows platform (or Linux); therefore, any clients you write must run on that type of machine. This isn't always desirable. For example, you might want to provide easy access to the data that exists on your database to anyone who has an Internet connection. Because you've already written an application server that acts as a broker for your data—in addition to housing business rules for that data—it would be desirable to reuse the application server as opposed to rewriting the entire data-access and business rule tier in another environment.

Straight HTML

This section focuses on how to leverage your application server while providing a new presentation tier that will use straight HTML. This section assumes that you're familiar with the material covered in Chapter 31, "Internet-Enabling Your Applications with WebBroker" of *Delphi 5 Developer's Guide*, which is on this book's CD-ROM. Using this method, you're introducing another layer into your architecture. WebBroker acts as the client to the application server and repackages this data into HTML that will be displayed on the browser. You also lose some of the benefits of working with the Delphi IDE, such as the lack of data-aware controls. However, this is a very viable option for allowing access to your data in a simple HTML format.

After creating a WebBroker Application and a `WebModule`, you simply place a `TDispatchConnection` and `TClientDataset` on the `WebModule`. Once the properties are filled in, you can use a number of different methods to translate this data into HTML that will eventually be seen by the client.

One valid technique would be to add a `TDatasetTableProducer` linked to the `TClientDataset` of interest. From there, the user can click a link and go to an edit page, where she can edit the data and apply the updates. See Listings 21.7 and 21.8 for a sample implementation of this technique.

LISTING 21.7 HTML for Editing and Applying Updates

```
<form action="<#SCRIPTNAME>/updaterecord" method="post">
<b>EmpNo: <#EMPNO></b>
<input type="hidden" name="EmpNo" value=<#EMPNO>>
<table cellspacing="2" cellpadding="2" border="0">
<tr>
  <td>Last Name:</td>
  <td><input type="text" name="LastName" value=<#LASTNAME>></td>
</tr>
<tr>
  <td>First Name:</td>
  <td><input type="text" name="FirstName" value=<#FIRSTNAME>></td>
```

LISTING 21.7 Continued

```
</tr>
<tr>
  <td>Hire Date:</td>
  <td><input type="text" name="HireDate" size="8" value=<#HIREDATE>></td>
</tr>
<tr>
  <td>Salary:</td>
  <td><input type="text" name="Salary" size="8" value=<#SALARY>></td>
</tr>
<tr>
  <td>Vacation:</td>
  <td><input type="text" name="Vacation" size="4" value=<#VACATION>></td>
</tr>
</table>
<input type="submit" name="Submit" value="Apply Updates">
<input type="Reset">
</form>
```

LISTING 21.8 Code for Editing and Applying Updates

```
unit WebMain;

interface

uses
  Windows, Messages, SysUtils, Classes, HTTPApp, DBWeb, Db, DBClient,
  MConnect, DSProd;

type
  TWebModule1 = class(TWebModule)
    dcJoin: TDCOMConnection;
    cdsJoin: TClientDataSet;
    dstpJoin: TDataSetTableProducer;
    dsppJoin: TDataSetPageProducer;
    ppSuccess: TPageProducer;
    ppError: TPageProducer;
    procedure WebModuleBeforeDispatch(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure WebModule1waListAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure dstpJoinFormatCell(Sender: TObject; CellRow,
      CellColumn: Integer; var BgColor: THTMLBgColor;
      var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs,
      CellData: String);
```

LISTING 21.8 Continued

```
    procedure WebModule1waEditAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure dsppJoinHTMLTag(Sender: TObject; Tag: TTag;
      const TagString: String; TagParams: TStrings;
      var ReplaceText: String);
    procedure WebModule1waUpdateAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
  private
    { Private declarations }
    DataFields : TStrings;
  public
    { Public declarations }
  end;

var
  WebModule1: TWebModule1;

implementation

{$R *.DFM}

procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  with Request do
    case MethodType of
      mtPost: DataFields:=ContentFields;
      mtGet: DataFields:=QueryFields;
    end;
end;

function LocalServerPath(sFile : string = '') : string;
var
  FN: array[0..MAX_PATH- 1] of char;
  sPath : shortstring;
begin
  SetString(sPath, FN, GetModuleFileName(hInstance, FN, SizeOf(FN)));
  Result := ExtractFilePath( sPath ) + ExtractFileName( sFile );
end;

procedure TWebModule1.WebModule1waListAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  cdsJoin.Open;
  Response.Content :=  dstpJoin.Content;
end;
```

LISTING 21.8 Continued

```
procedure TWebModule1.dstpJoinFormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellRow > 0) and (CellColumn = 0) then
    CellData := Format('<a href="%s/getrecord?empno=%s">%s</a>',
      [Request.ScriptName, CellData, CellData]);
end;

procedure TWebModule1.WebModule1waEditAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  dsppJoin.HTMLFile := LocalServerPath('join.htm');
  cdsJoin.Filter := 'EmpNo = ' + DataFields.Values['empno'];
  cdsJoin.Filtered := true;
  Response.Content := dsppJoin.Content;
end;

procedure TWebModule1.dsppJoinHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if CompareText(TagString, 'SCRIPTNAME')=0 then
   ReplaceText:=Request.ScriptName;
end;

procedure TWebModule1.WebModule1waUpdateAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  EmpNo, LastName, FirstName, HireDate, Salary, Vacation:  string;
begin
  EmpNo:=DataFields.Values['EmpNo'];
  LastName:=DataFields.Values['LastName'];
  FirstName:=DataFields.Values['FirstName'];
  HireDate:=DataFields.Values['HireDate'];
  Salary:=DataFields.Values['Salary'];
  Vacation:=DataFields.Values['Vacation'];

  cdsJoin.Open;
  if cdsJoin.Locate('EMPNO', EmpNo, []) then
  begin
    cdsJoin.Edit;
    cdsJoin.FieldByName('LastName').AsString:=LastName;
    cdsJoin.FieldByName('FirstName').AsString:=FirstName;
    cdsJoin.FieldByName('HireDate').AsString:=HireDate;
```

LISTING 21.8 Continued

```
    cdsJoin.FieldByName('Salary').AsString:=Salary;
    cdsJoin.FieldByName('Vacation').AsString:=Vacation;
    if cdsJoin.ApplyUpdates(0)=0 then
      Response.Content:=ppSuccess.Content else
      Response.Content:=pPError.Content;
  end;
end;

end.
```

Note that this method requires much custom code to be written, and the full feature set of DataSnap isn't implemented in this example—specifically error reconciliation. You can continue to enhance this example to be more robust if you use this technique extensively.

CAUTION

It's imperative that you consider the concept of state when writing your `WebModule` and application server. Because HTTP is a stateless protocol, you cannot rely on the values of properties to be the same as you left them when the call was over.

TIP

WebBroker is one way to get your data to Web browsers. Using WebSnap, you can extend the capabilities of your application even further by using the new features WebSnap offers, such as scripting and session support.

To run this sample, be sure to compile and register the Join2 sample application. Next, compile the Web application (either the CGI or ISAPI version), and place the executable in a script-capable directory for your Web server. The code also expects to find the file `join.htm` in the scripts directory, so copy that too. Then, just point your browser to `http://localhost/scripts/WebJoin.exe` and see the results of this sample.

You'll find an example on the book's CD-ROM in the directory for this chapter under `\WebBrok`.

InternetExpress

With InternetExpress, you can enhance the functionality of a straight `WebModule` approach to allow for a richer experience on the client. This is possible due to the use of open standards such as XML and JavaScript in InternetExpress. Using InternetExpress, you can create a browser-only front end to your DataSnap application server: no ActiveX controls to download; zero client-side install and configuration requirements; nothing but a Web browser hitting a Web server.

In order to use InternetExpress, you will need to have some code running on a Web server. For this example, we will use an ISAPI application, but you could also use CGI or ASP. The purpose of the Web broker is to take requests from the browser and pass those requests on to the app server. Placing InternetExpress components in the Web broker application makes this task very easy.

This example will use a standard DataSnap app server that has Customers, Orders, and Employees. Customers and Orders are linked in a nested dataset relationship (for more information on nested datasets, see the next section), whereas the Employees dataset will serve as a lookup table. See the accompanying source code for the app server definition. After the app server has been built and registered, you can focus on building the Web broker application that will communicate with the app server.

Create a new ISAPI application by selecting File, New, Web Server Application from the Object Repository. Place a `TDCOMConnection` component on the `WebModule`. This will act as the link to the app server, so fill in the `ServerName` property with the ProgID of the app server.

Next, you will place a `TXMLBroker` component from the InternetExpress page of the Component Palette on the WebModule and set the `RemoteServer` and `ProviderName` properties to the CustomerProvider. The `TXMLBroker` component acts in a manner similar to the `TClientDataset`. It is responsible for retrieving data packets from the app server and passing those data packets to the browser. The main difference between the data packet in a `TXMLBroker` and a `TClientDataset` is that the `TXMLBroker` translates the DataSnap data packets into XML. You will also add a `TClientDataset` to the `WebModule` and tie it to the Employees provider on the app server. You will use this as a lookup datasource later.

The `TXMLBroker` component is responsible for communication to the application server and also the navigation of HTML pages. Many properties are available to customize how your InternetExpress application will behave. For example, you can limit the number of records that will be transmitted to the client or specify the number of errors allowed during an update.

You now need a way to move this data to the browser. Using the `TInetXPageProducer` component, you can use the WebBroker technology in Delphi to serve an HTML page up to the browser. However, the `TInetXPageProducer` also allows for visual creation of the Web page via the Web Page Editor.

Double-click on the `TInetXPageProducer` to bring up the Web Page Editor. This visual editor helps you customize what elements are present on a given Web page. One of the most interesting things about InternetExpress is that it is completely extensible. You can create your own components that can be used in the Web Page Editor by following some well-defined rules. For examples of custom InternetExpress components, see the `<DELPHI>\DEMOS\MIDAS\INTERNETEXPRESS\INETXCUSTOM` directory.

CAUTION

`TInetXPageProducer` has a property named `IncludePathURL`. It is essential to set this property properly, or your InternetExpress application won't work. Set the value to the virtual directory that contains the InternetExpress JavaScript files. For example, if you place the files in `c:\inetpub\wwwroot\jscript`, the value for this property will be `/jscript/`.

With the Web Page Editor active, click the Insert tool button to display the Add Web Component dialog box (see Figure 21.7). This dialog box contains a list of Web components that can be added to the HTML page. This list is based on which parent component (the section in the upper left) is currently selected. For example, add a DataForm Web component to the root node to allow end users to display and edit database information in a form-like layout.

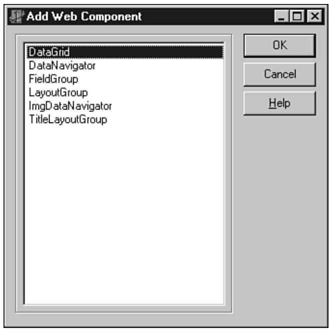


FIGURE 21.7
The Add Web Component dialog box from the Web Page Editor.

If you then select the DataForm node in the Web Page Editor, you can click the Insert button again. Notice that the list of components available at this point is different from the list displayed from the previous step. After selecting the FieldGroup component, you will see a warning in the preview pane, telling you that the `TXMLBroker` property for the FieldGroup isn't assigned. By assigning the XMLBroker in the Object Inspector, you will immediately notice the layout of the HTML in the preview pane of the Web Page Editor. As you continue to modify properties or add components, the state of the HTML page will be constantly updated (see Figure 21.8).

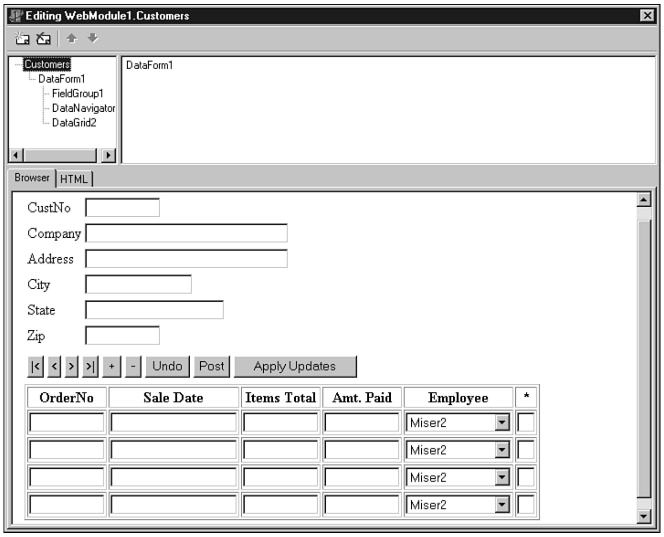


FIGURE 21.8
The Web Page Editor after designing an HTML page.

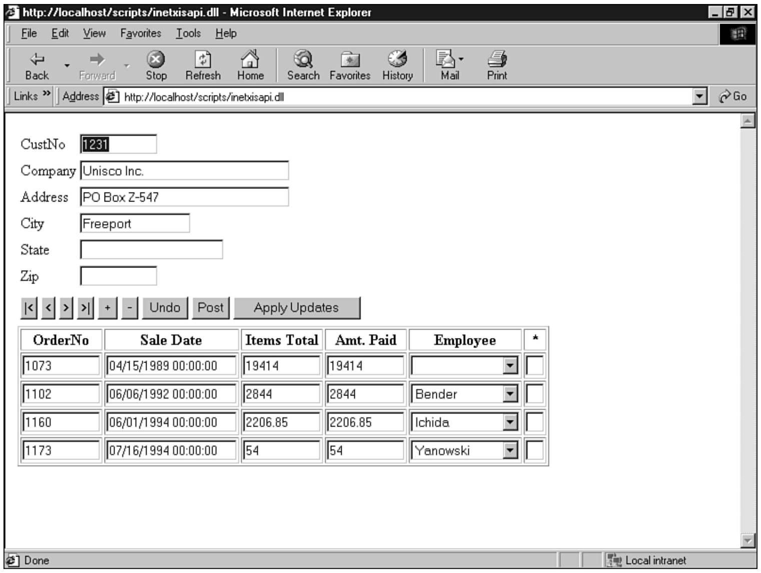
The level of customization available with the standard Web components is practically limitless. Properties make it easy to change field captions, alignment, colors; add straight custom HTML code; and even use style sheets. Furthermore, if the component doesn't suit your needs exactly, you can always create a descendant component and use that in its place. The framework is truly as extensible as your imagination allows.

In order to call the ISAPI DLL, you need to place it in a virtual directory capable of executing script. You also need to move the JavaScript files found in `<DELPHI>\SOURCE\WEBMIDAS` to a valid location on your Web server and modify the `TInetXPageProducer.IncludePathURL` property to point to the URI of the JavaScript files. After that, the page is ready to be viewed.

To access the page, all you need is a JavaScript-capable browser. Simply point the browser to `http://localhost/inetx/inetxisapi.dll`, and the data will display in the browser. Figure 21.9 shows a screenshot of the application in action.

You can detect reconciliation errors during the ApplyUpdates process as you are already used to doing in a standalone DataSnap application. This capability is made possible when you assign the `TXMLBroker.ReconcileProducer` property to a `TPageProducer`. Whenever an error occurs, the `Content` of the `TPageProducer` assigned to this property will be returned to the end user.

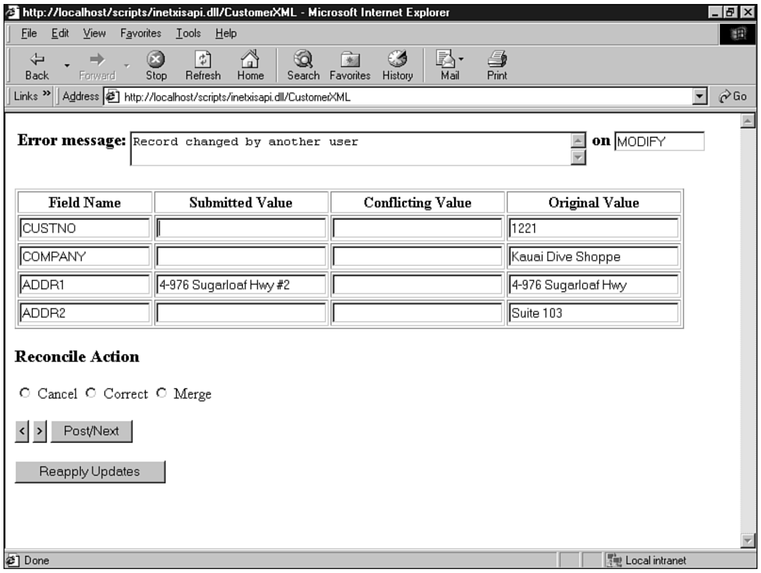
A specialized `TPageProducer`, `TReconcilePageProducer`, is available by installing the `InetXCustom.dpk` package found in `<DELPHI>\DEMOS\MIDAS\INTERNETEXPRESS\INETXCUSTOM`. This PageProducer generates HTML that acts much like the standard DataSnap Reconciliation Error dialog box (see Figure 21.10).


http://localhost/scripts/inetxisapi.dll - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://localhost/scripts/inetxisapi.dll Go
|< < > >| + - Undo Post Apply Updates
Done
Local intranet

CustNo	1231
Company	Unisco Inc.
Address	PO Box Z-547
City	Freeport
State	
Zip	

OrderNo	Sale Date	Items Total	Amt. Paid	Employee	*
1073	04/15/1989 00:00:00	19414	19414		
1102	06/06/1992 00:00:00	2844	2844	Bender	
1160	06/01/1994 00:00:00	2206.85	2206.85	Ichida	
1173	07/16/1994 00:00:00	54	54	Yanowski	

FIGURE 21.9
Internet Explorer accessing the InternetExpress Web page.


http://localhost/scripts/inetxisapi.dll/CustomerXML - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://localhost/scripts/inetxisapi.dll/CustomerXML Go
< > Post/Next
Reapply Updates
Done
Local intranet

Error message: Record changed by another user **on** MODIFY

Field Name	Submitted Value	Conflicting Value	Original Value
CUSTNO			1221
COMPANY			Kauai Dive Shoppe
ADDR1	4-976 Sugarloaf Hwy #2		4-976 Sugarloaf Hwy
ADDR2			Suite 103

Reconcile Action

○ Cancel ○ Correct ○ Merge

FIGURE 21.10
View of the HTML page generated by `TReconcilePageProducer`.

You'll find an example on the book's CD-ROM in the directory for this chapter under `\InetX`.

More Client Dataset Features

Many options are available to control the TClientDataset component. In this section, we will look at ways to use the TClientDataset to make coding easier in complex applications.

Two-Tier Applications

You've seen how to assign the provider—and therefore the data—to the ClientDataset in a three-tier application. However, many times a simple two-tier application is all that's needed. So, how do you use DataSnap in a two-tier application? There are four possibilities:

- Runtime assignment of data
- Design-time assignment of data
- Runtime assignment of a provider
- Design-time assignment of a provider

The two basic choices when using ClientDataset are assigning the AppServer property and assigning the data. If you choose to assign the AppServer, you have a link between the TDatasetProvider and the ClientDataset that will allow you to have communication between the ClientDataset and TDatasetProvider as needed. If, on the other hand, you choose to assign the data, you have effectively created a local storage mechanism for your data and the ClientDataset will not communicate with the TDatasetProvider component for more information or data.

In order to assign the data directly from a TDataset to a TClientDataset at runtime, use the code in Listing 21.9.

LISTING 21.9 Code to Assign Data Directly from a TDataSet

```
function GetData(ADataset: TDataset): OleVariant;
begin
  with TDatasetProvider.Create(nil) do
  try
    Dataset:=ADataset;
    Result:=Data;
  finally
    Free;
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  ClientDataset1.Data:=GetData(ADOTable1);
end;
```

This method takes more code and effort than previous versions of Delphi, where you would simply assign the `Table1.Provider.Data` property to the `ClientDataset1.Data` property. However, this function will help make the additional code less noticeable.

You can also use the `TClientDataset` component to retrieve the data from a `TDataset` at design time by selecting the Assign Local Data command from the context menu of the `TClientDataset` component. Then, you specify the `TDataset` component that contains the data you want, and the data is brought to the `TClientDataset` and stored in the `Data` property.

CAUTION

If you were to save the file in this state and compare the size of the DFM file to the size before executing this command, you would notice an increase in the DFM size. This is because Delphi has stored all the metadata and records associated with the `TDataset` in the DFM. Delphi will only stream this data to the DFM if the `TClientDataset` is `Active`. You can also trim this space by executing the Clear Data command on the `TClientDataset` context menu.

If you want the full flexibility that a provider assignment allows, you need to assign the `AppServer` property. At runtime, you can assign the `AppServer` property in code. This can be as simple as the following statement, found in `FormCreate`:

```
ClientDataset1.AppServer:=TLocalAppServer.Create(Table1);
ClientDataset1.Open;
```

You can assign the `AppServer` property at design time. If you leave the `RemoteServer` property blank on a `TClientDataset`, you can assign a `TDatasetProvider` component to the `TClientDataset.ProviderName` property.

One major drawback to using the `TClientDataset.ProviderName` property is that it can't be assigned to providers that reside on another form or `DataModule` at design time. This is why Delphi 6 introduced the `TLocalConnection` component. `TLocalConnection` will autodiscover and expose any `TDatasetProviders` that it finds with the same owner. To use this method of assigning providers, assign the `ClientDataset.RemoteServer` property to be the `LocalConnection` component on the external form or `DataModule`. After doing this, you will have the list of providers for that `LocalConnection` in the `ClientDataset.ProviderName` property.

The major difference between using `TDataset` components and `ClientDataset` is that when you're using `ClientDataset`, you're using the `IAppServer` interface to broker your requests for data to the underlying `TDataset` component. This means that you'll be manipulating the properties, methods, events, and fields of the `TClientDataset` component, not the `TDataset` component. Think of the TDataset component as if it were in a separate application and therefore can't be manipulated directly by you in code. Place all your server components on a separate DataModule. Placing the TDatabase, TDataset, and TLocalConnection components on a separate DataModule effectively prepares your application for an easier transition to a multitier deployment later on. Another benefit of doing this is that it might help you think of the DataModule as something that the client cannot touch easily. Again, this is good preparation for your application, and your own mindset, when it comes time to port this application to a multitier deployment.

Classic Mistakes

The most common mistake in creating a multitier application is introducing unnecessary knowledge of the data tier into the presentation tier. Some validation is more suitable in the presentation tier, but it's how that validation is performed that determines its suitability in a multitier application.

For example, if you're passing dynamic SQL statements from the client to the server, this introduces a dependency for the client application to always be synchronized with the data tier. Doing things this way introduces more moving parts that need to be coordinated in the overall multitier application. If you change one of the tables' structures on the data tier, you must update all the client applications that send dynamic SQL so that they can now send the proper SQL statement. This clearly limits the benefit that a properly developed thin-client application holds.

Another example of a classic mistake is when the client application attempts to control the transaction lifetime, as opposed to allowing the business tier to take care of this on the client's behalf. Most of the time, this is implemented by exposing three methods of the TDataBase instance on the server—BeginTransaction(), Commit(), and Rollback()—and calling those methods from the client. Doing things in this manner makes the client code much more complicated to maintain and violates the principle that the presentation tier should be the only tier responsible for communication to the data tier. The presentation tier should never have to rely on such an approach. Instead, you should send your updates to the business tier and let that tier deal with updating the data in a transaction.

Deploying DataSnap Applications

After you've built a complete DataSnap application, the last hurdle left to clear is deploying that application. This section outlines what needs to be done in order to make your DataSnap application deployment painless.

Licensing Issues

Licensing has been a tough subject for many people ever since DataSnap was first introduced in Delphi 3. The myriad of options for deploying this technology has contributed to this confusion. This section details the overall requirements of when you need to purchase a DataSnap license. However, the only legally binding document for licensing is in `DEPLOY.TXT`, located in the Delphi 6 directory. Finally, for the ultimate authority to answer this question for a specific situation, you must contact your local Borland sales office. More guidelines and examples are available at

`http://www.borland.com/midas/papers/licensing/`

or our Web site at

`http://www.xapware.com/ddg`

The information from this document was prepared to answer some of the more common scenarios in which DataSnap is used. Pricing information and options are also included in the document.

The key criteria to determine the necessity of a DataSnap license for your application is whether the DataSnap data packet crosses a machine boundary. If it does, and you use the DataSnap components on both machines, you need to purchase a license. If it doesn't (as in the one- and two-tier examples presented earlier), you're using DataSnap technology, but there's no need to purchase a license to use DataSnap in this manner.

DCOM Configuration

DCOM configuration appears to be as much art as it is science. There are many aspects to a complete and secure DCOM configuration, but this section will help you understand some of the basics of this black art.

After registering your application server, your server object is now available for customization in the Microsoft utility DCOMCNFG. This utility is included with NT systems automatically but is a separate download for Win9*x* machines. As a side note, there are plenty of bugs in DCOMCNFG; the most notable being DCOMCNFG can only be run on Win9*x* machines that have User-level share enabled. This, of course, requires a domain. This isn't always possible or desirable in a peer-to-peer network, such as two Windows 9*x* machines. This has led many people to incorrectly assume that an NT machine is required to run DCOM.

If you can run DCOMCNFG, you can select the registered application server and click the Properties button to reveal information about your server. The Identity page is a good place to start in our brief tour of DCOMCNFG. The default setting for a registered server object is Launching User. Microsoft couldn't have made a worse decision for the default if it tried.

When DCOM creates the server, it uses the security context of the user specified on the Identity page. The launching user will spawn one new process of the server object for each and every distinct user login. Many people look at the fact that they selected the `ciMultiple` instancing mode and wonder why multiple copies of their server are being created. For example, if user A connects to the server and then user B connects, DCOM will spawn an entirely new process for user B. Additionally, you won't see the GUI portion of the server for users who log in under an account different from that currently in use on the server machine. This is because of the NT concept known as *Windows stations*. The only Windows station capable of writing to the screen is the Interactive User, which is the user who is currently logged in on the server machine. Furthermore, windows stations are a scarce resource, and you might not be able to run many server processes if you use this setting. In summary, never use the Launching User option as your identity for your server.

The next interesting option on this page is the Interactive User, which means that every single client that creates a server will do so under the context of the user who is logged in to the server at that point in time. This will also allow you to have visual interaction with your application server. Unfortunately, most system administrators don't allow an open login to just sit there idle on an NT machine. In addition, if the logged-in user decides to log out, the application server will no longer work as desired.

For this discussion, this only leaves the last enabled option on the Identity page: This User. Using this setting, all clients will create one application server and use the login credentials and context of the user specified on the Identity page. This also means that the NT machine doesn't require a user to be logged in to use the application server. The one downside to this approach is that there will be no GUI display of the server when using this option. However, it is by far the best of all available options to put your application server in production.

After the server object is configured properly with the right identity, you need to turn your attention to the Security tab. Make sure that the user who will be running this object has the appropriate privileges assigned. Also be sure to grant the SYSTEM user access to the server; otherwise, you'll encounter errors along the way.

Many subtle nuances are strewn throughout the DCOM configuration process. For the latest on DCOM configuration issues, especially as they pertain to Windows 9*x*, Delphi, and DataSnap, visit the DCOM page of our Web site at

`http://www.DistribuCon.com/dcom95.htm`

Files to Deploy

The requirements for deploying a DataSnap application have changed with each new release of Delphi. Delphi 6 makes deployment easier than any other version.

With Delphi 6, the minimum files needed for deployment of your DataSnap application is shown in the following lists.

Here are the steps for the server (these steps assume a COM server; they will differ slightly for other varieties):

1. Copy the application server to a directory with sufficient NTFS privileges or share level privileges set properly if on a Win9x machine.
2. Install your data access layer to allow the application server to act as a client to the RDBMS (for example, BDE, MDAC, specific client-side database libraries, and so on).
3. Copy `MIDAS.DLL` to the `%SYSTEM%` directory. By default, this would be `C:\Winnt\System32` for NT machines and `C:\Windows\System` for 9*x* machines.
4. Run the application server once to register it with COM.

Here are the steps for the client:

1. Copy the client to a directory, along with any other external dependency files used by your client (for example, runtime packages, DLLs, ActiveX controls, and so on).
2. Copy `MIDAS.DLL` to the `%SYSTEM%` directory. Note that Delphi 6 can statically link `MIDAS.DLL` into your application, thus making this step unnecessary. To do this, simply add the unit MidasLib to your uses clause and rebuild your application. You will see an increase in the size of the EXE due to the static linking.
3. Optional: If you specify the `ServerName` property in your `TDispatchConnection` or if you employ early binding in your client, you need to register the server's type library (TLB) file. This can be done by using a utility such as `<DELPHI>\BIN\TREGSVR.EXE` (or programmatically if you so choose).

Internet Deployment Considerations (Firewalls)

When deploying your application over a LAN, there's nothing to get in your way. You can choose whatever connection type best suits your application's needs. However, if you need to rely on the Internet as your backbone, many things can go wrong—namely, firewalls.

DCOM isn't the most firewall-friendly protocol. It requires opening multiple ports on a firewall. Most system administrators are wary of opening an entire range of ports (particularly those commonly recognized as DCOM ports) because it invites hackers to come knocking on the door. Using `TSocketConnection`, the story improves somewhat. The firewall only needs one open port. However, the occasional system administrator will even refuse to do that on the grounds that this is a security breach.

`TWebConnection` is similar to `TSocketConnection` in that it permits DataSnap traffic to be bundled up into valid HTTP traffic, and then uses the most open port in the world—the HTTP port (default port 80). Actually, the component even supports SSL, so you can have secure communications. By doing this, all firewall issues are completely eliminated. After all, if a corporation doesn't allow HTTP traffic in or out, nothing can be done to communicate with them anyway.

This bit of magic is accomplished by using the Borland-provided ISAPI extension that translates HTTP traffic into DataSnap traffic, and vice versa. In this regard, the ISAPI DLL does the same work that ScktSrvr does for socket connections. The ISAPI extension `httpsrvr.dll` needs to be placed in a directory capable of executing code. For example, with IIS, the default location for this file would be in `C:\Inetpub\Scripts`.

One more benefit of using `TWebConnection` is that it supports object pooling. Object pooling is used to spare the server the overhead of object creation every time a client connects to the server. Furthermore, the pooling mechanism in DataSnap allows for a maximum number of objects to be created. After this maximum has been reached, an error will be sent to the client saying that the server is too busy to process this request. This is more flexible and scalable than just creating an arbitrary number of threads for every single client that wants to connect to the server.

To take this a step further, building your RDM as a Web Service using a SOAP Data Module will not only provide the benefits of a `TwebConnection`, but will also permit clients using industry-standard SOAP protocols to be constructed. This platform enables your application server for use by .Net, Sun ONE, and other industry-compliant SOAP systems.

In order to tell DataSnap that this RDM will be pooled, you need to call `RegisterPooled` and `UnregisterPooled` in the `UpdateRegistry` method of the RDM. (See Listing 21.1 for a sample implementation of `UpdateRegistry`.) The following is a sample call to the `RegisterPooled` method:

```
RegisterPooled(ClassID, 16, 30);
```

This call tells DataSnap that 16 objects will be available in the pool, and that DataSnap can free any instances of objects that have been created if there has been no activity for 30 minutes. If you never want to free the objects, then you can pass `0` as the timeout parameter.

The client doesn't change that drastically. Simply use a `TWebConnection` as the `TDispatchConnection` for the client and fill in the appropriate properties, and the client will be communicating to the application server over HTTP. The one major difference when using `TWebConnection` is the need to specify the complete URL to the `httpsrvr.dll`, as opposed to just identifying the server computer by name or address. Figure 21.11 shows a screenshot of a typical setup using `TWebConnection`.

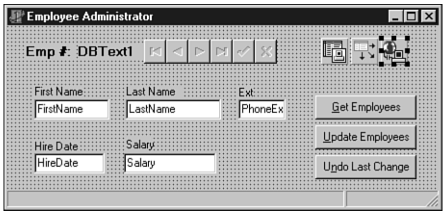


FIGURE 21.11
`TWebConnection` *setup at design time.*

Another benefit of using HTTP for your transport is that an OS such as NT Enterprise allows you to cluster servers. This provides automated load balancing and fault tolerance for your application server. For more information about clustering, see `http://www.microsoft.com/ntserver/ntserverenterprise/exec/overview/clustering`.

The limitations of using `TWebConnection` are fairly trivial, and they're well worth any concession in order to have more clients capable of reaching your application server. The limitations are that you must install `wininet.dll` on the client, and no callbacks are available when using `TWebConnection`.

Summary

This chapter provides quite a bit of information on DataSnap. Still, it only scratches the surface of what can be done with this technology—something far beyond the scope of a single chapter. Even after you explore all the nooks and crannies of DataSnap, you can still add to your knowledge and capabilities by using DataSnap with C++Builder and JBuilder. Using JBuilder, you can achieve the nirvana of cross-platform access to an application server while using the same technology and concepts you learned here.

DataSnap is a fast-evolving technology that brings the promise of multitier applications to every programmer. Once you experience the true power of creating an application with DataSnap, you might never return to database application development as you know it today.