Client Tracker: MIDAS Development

CHAPTER

34

IN THIS CHAPTER

In the previous chapter, “Inventory Manager: Client/Server Development,” we discussed techniques for developing two-tier applications. In this chapter, we will create a three-tier application using the MIDAS technology presented in Chapter 32, “MIDAS Development.” The focus of this chapter is to illustrate the simplicity of using the MIDAS components as well as the briefcase model for offsite work.

The application we will develop lends itself to the briefcase model. This application is a client tracker or manager. Often, sales reps perform much of their work offsite, possibly travelling to several different locations. The client list that these sales reps work with might be critical to both the reps and their parent company. Therefore, this client information should probably reside at the company site. How then, can the sales rep make use of this data without having to rely on a network connection? Also, how can the sales rep update the company data with newer information that he or she is likely to get at the client’s site?

`TClientDataSet` makes it possible to create briefcase applications with its implementation of *internal caching,* which allows the sales reps to download the data or even a subset of the data with which they can work offsite. Later, when they return to home base, they can upload any changes made to the database. In this chapter, we will build a simple client management tool that illustrates this approach to building briefcase applications.

Designing the Server Application

The server application is designed using the same procedure discussed back in Chapter 32. Here, you will see our `TRemoteDataModule`, called `CustomerRemoteDataModule`, containing `TSession`, `TDataBase`, `TQuery`, and `TDataSetProvider` components. The `TSession` component, `ssnCust`, is provided to handle multi-instancing issues (its `AutoSessionName` property is set to `True`). `DbCust`, the `TDataBase` component, provides the client connection to the database and prevents the login dialog from displaying. `QryCust`, the `TQuery` component, returns the result set to the client table. `PrvCust` is bound to `qryCust` through its `DataSet` property. We use the same Customer table presented in the last chapter.

Listing 34.1 shows the source code for the remote data module.

LISTING 34.1 Customer Remote Data Module Source Code

```
unit CustRDM;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ComServ, ComObj, VCLCom, StdVcl, DataBkr, DBClient, CustServ_TLB,
  Db, DBTables, Provider;
```

```
type

  TFilterType = (ftNone, ftCity, ftState);

  TCustomerRemoteDataModule = class(TRemoteDataModule,
     ICustomerRemoteDataModule)
    ssnCust: TSession;
    dbCust: TDatabase;
    qryCust: TQuery;
    prvCust: TDataSetProvider;
  private
    FFilterStr: String;
    FFilterType: TFilterType;
  public
    { Public declarations }
  protected
    procedure FilterByCity(const ACity: WideString; out Data: OleVariant);
      safecall;
    procedure FilterByState(const AStateStr: WideString; out Data: OleVariant);
      safecall;
    procedure NoFilter(out Data: OleVariant); safecall;
  end;

var
  CustomerRemoteDataModule: TCustomerRemoteDataModule;

implementation

{$R *.DFM}

procedure TCustomerRemoteDataModule.FilterByCity(const ACity: WideString;
  out Data: OleVariant);
begin
  FFilterType  := ftCity;
  FFilterStr   := ACity;
  qryCust.Close;
  qryCust.SQL.Clear;
  qryCust.SQL.Add(Format('select * from CUSTOMER where CITY = "%s"', [ACity]));
  qryCust.Open;
  Data := prvCust.Data;
end;

procedure TCustomerRemoteDataModule.FilterByState(
  const AStateStr: WideString; out Data: OleVariant);
begin
  FFilterType  := ftState;
```

continues

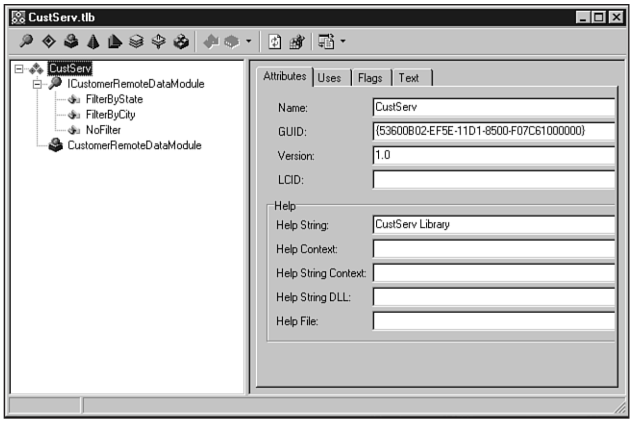
LISTING 34.1 Customer Remote Data Module Source Code

```
  FFilterStr   := AStateStr;
  qryCust.Close;
  qryCust.SQL.Clear;
  qryCust.SQL.Add(Format('select * from CUSTOMER where STATE = "%s"',
    [AStateStr]));
  qryCust.Open;
  Data := prvCust.Data;
end;

procedure TCustomerRemoteDataModule.NoFilter(out Data: OleVariant);
begin
  FFiltertype := ftNone;
  qryCust.Close;
  qryCust.SQL.Clear;
  qryCust.SQL.Add('select * from CUSTOMER');
  qryCust.Open;
  Data := prvCust.Data;
end;

initialization
  TComponentFactory.Create(ComServer, TCustomerRemoteDataModule,
    Class_CustomerRemoteDataModule, ciMultiInstance, tmSingle);
end.
```

Listing 34.1 shows three methods that were added to `TCustomerDataModule`. These methods were actually added to the interface `ICustomerRemoteDateModule` through the Type Library Editor, which in turn created the implementation methods for the `TCustomerRemoteDataModule` class (see Figure 34.1). In the Type Library Editor, we defined the methods and their parameters and then added the code to each implementation method created by Delphi. You can examine the source code for the type library in the file `CustServ_TLB.pas`.

NOTE

This demo is a conversion of a demo written for Delphi 4. To port Delphi 4 MIDAS servers to Delphi 5, you must perform a few steps. These steps are detailed in the online help under “Converting MIDAS Applications.”



CustServ
- ICustomerRemoteDataModule
  - FilterByState
  - FilterByCity
  - NoFilter
- CustomerRemoteDataModule

Name:	CustServ
GUID:	{53600B02-EF5E-11D1-8500-F07C61000000}
Version:	1.0
LCID:	

Help

Help String:	CustServ Library
Help Context:	
Help String Context:	
Help String DLL:	
Help File:	

FIGURE 34.1
The Type Library Editor.

The methods `FilterByCity()` and `FilterByState()` are used to allow the client to download a subset of the entire table. This makes sense, because it might not be necessary to download the entire client list when the sales rep is traveling to a single location. Both of these methods take a string parameter that is used to specify the filter value. `NoFilter()` removes any filtering applied to the table.

These methods cause server-side filtering in that they provide a way to limit the records returned to the client. Alternatively, the user may want to perform filtering on the client side. That is, the sales rep may want to access the entire result set but have the ability to filter out desired records as needed. We will illustrate both techniques.

Designing the Client Application

The client application contains a data module and a main form. We will discuss the data module first.

Client Data Module

The data module for the Client Tracker application illustrates several techniques. First, it illustrates how to implement the briefcase model. Second, it shows how to make its mode (online/offline) persistent. In other words, when the user shuts down the application, it will recall its state when executed again. This prevents the application from attempting to attach to the server when the client is running it offline. We also illustrate how to perform client-side filtering. When the user is online, the application will perform server-side filtering. When the user is offline, filtering is performed on the client end. Listing 34.2 shows the source code for `CustomerDataModule`.

LISTING 34.2 Customer Data Module Source Code

```
unit CustDM;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBClient, MConnect, Db;

const
  cFileName        = 'CustData.cds';

  cRegIniFile      = 'Software\DDG Client App';
  cRegSection      = 'Startup Config';
  cRegOnlineIdent  = 'Run Online';

type

  TFilterType = (ftNone, ftByCity, ftByState);

  TAddErrorToClientEvent = procedure(const AFieldName, OldStr, NewStr,
    CurStr, ErrMsg: String) of Object;


  TCustomerDataModule = class(TDataModule)
    cdsCust: TClientDataSet;
    dcomCust: TDCOMConnection;
    cdsCustCUSTOMER_ID: TIntegerField;
    cdsCustFNAME: TStringField;
    cdsCustLNAME: TStringField;
    cdsCustCREDIT_LINE: TSmallintField;
    cdsCustWORK_ADDRESS: TStringField;
    cdsCustALT_ADDRESS: TStringField;
    cdsCustCITY: TStringField;
    cdsCustSTATE: TStringField;
    cdsCustZIP: TStringField;
    cdsCustWORK_PHONE: TStringField;
    cdsCustALT_PHONE: TStringField;
    cdsCustCOMMENTS: TMemoField;
    cdsCustCOMPANY: TStringField;
    procedure CustomerDataModuleCreate(Sender: TObject);
    procedure cdsCustReconcileError(DataSet: TClientDataSet;
      E: EReconcileError; UpdateKind: TUpdateKind;
      var Action: TReconcileAction);
    procedure CustomerDataModuleDestroy(Sender: TObject);
    procedure cdsCustFilterRecord(DataSet: TDataSet; var Accept: Boolean);
  private
```

```
    FFilterType: TFilterType;
    FFilterStr: String;
    FOnAddErrorToClient: TAddErrorToClientEvent;

    function GetOnline: Boolean;
    procedure SetOnline(const Value: Boolean);
    { Private declarations }
  protected
    function GetChangeCount: Integer;

  public
    procedure EditClient;
    procedure AddClient;
    procedure SaveClient;
    procedure CancelClient;
    procedure DeleteClient;
    procedure ApplyUpdates;
    procedure CancelUpdates;
    procedure First;
    procedure Previous;
    procedure Next;
    procedure Last;
    function IsBOF: Boolean;
    function IsEOF: Boolean;

    procedure FilterByState;
    procedure FilterByCity;
    procedure NoFilter;

    property ChangeCount: Integer read GetChangeCount;
    property Online: Boolean read GetOnline write SetOnline;

    property OnAddErrorToClient: TAddErrorToClientEvent
       read FonAddErrorToClient
      write FOnAddErrorToClient;

  end;

var
  CustomerDataModule: TCustomerDataModule;

implementation
uses MainCustFrm, Registry;

{$R *.DFM}
```

continues

LISTING 34.2 Continued

```
procedure TCustomerDataModule.AddClient;
begin
  cdsCust.Insert;
end;

procedure TCustomerDataModule.ApplyUpdates;
begin
  cdsCust.ApplyUpdates(-1);
end;

procedure TCustomerDataModule.CancelClient;
begin
  cdsCust.Cancel;
end;

procedure TCustomerDataModule.CancelUpdates;
begin
  cdsCust.CancelUpdates;
end;

procedure TCustomerDataModule.DeleteClient;
begin
  if MessageDlg('Are you sure you want to delete the current record?',
  mtConfirmation, [mbYes, mbNo], 0) = mrYes then
      cdsCust.Delete;
end;

procedure TCustomerDataModule.EditClient;
begin
  cdsCust.Edit;
end;

function TCustomerDataModule.IsBOF: Boolean;
begin
  Result := cdsCust.Bof;
end;

function TCustomerDataModule.IsEOF: Boolean;
begin
  Result := cdsCust.Eof;
end;

procedure TCustomerDataModule.First;
begin
  cdsCust.First;
```

```
end;

procedure TCustomerDataModule.Last;
begin
  cdsCust.Last;
end;

procedure TCustomerDataModule.Next;
begin
  cdsCust.Next;
end;

procedure TCustomerDataModule.Previous;
begin
  cdsCust.Prior;
end;

procedure TCustomerDataModule.SaveClient;
begin
  cdsCust.Post;
end;

procedure TCustomerDataModule.cdsCustReconcileError(
  DataSet: TClientDataSet; E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
{ If an error occurs, update the appropriate TListview on the main form with
  the error data.  }
var
  CurStr, NewStr, OldStr: String;
  i: integer;
  V: Variant;

procedure SetString(V: Variant; var S: String);
{ We must test for a null value on V which would be returned if the
  table field was null. This is necessary because we can't typecast null
  as a string. }
begin

  if VarIsNull(V) then
    S := EmptyStr
  else
    S := String(V);
end;

begin
  for i := 0 to DataSet.FieldCount - 1 do
```

continues

LISTING 34.2 Continued

```
  begin

    V := DataSet.Fields[i].NewValue;
    SetString(V, NewStr);

    V := DataSet.Fields[i].CurValue;
    SetString(V, CurStr);

    V := DataSet.Fields[i].OldValue;
    SetString(V, OldStr);

    if NewStr <> CurStr then
      if Assigned(FOnAddErrorToClient) then
        FOnAddErrorToClient(DataSet.Fields[i].FieldName, OldStr, NewStr,
        CurStr, E.Message)
  end;
  //Update record and removes changes from the change log.
  Action := raRefresh;
end;

function TCustomerDataModule.GetChangeCount: Integer;
begin
  Result := cdsCust.ChangeCount;
end;

function TCustomerDataModule.GetOnline: Boolean;
begin
  Result := dcomCust.Connected;
end;

procedure TCustomerDataModule.SetOnline(const Value: Boolean);
begin

 if Value = True then
 begin
   dcomCust.Connected := True;

   if cdsCust.ChangeCount > 0 then begin
     ShowMessage('Your changes must be applied before going online');
     cdsCust.ApplyUpdates(-1);
   end;
   cdsCust.Refresh;

 end
 else begin
```

```
    cdsCust.FileName := cFileName;
    dcomCust.Connected := False;
  end;
end;

procedure TCustomerDataModule.CustomerDataModuleCreate(Sender: TObject);
{ Determine if the user last left the application online or offline and
  re-launch the application in that same mode. }
var
  RegIniFile: TRegIniFile;
  IsOnline: Boolean;
begin
  RegIniFile := TRegIniFile.Create(cRegIniFile);
  try
    IsOnline := RegIniFile.ReadBool(cRegSection, cRegOnlineIdent, True);
  finally
    RegIniFile.Free;
  end;

  if IsOnline then
  begin
    dcomCust.Connected := True;
    cdsCust.Open;
  end
  else begin
    cdsCust.FileName := cFileName;
    cdsCust.Open;
  end;

end;

procedure TCustomerDataModule.CustomerDataModuleDestroy(Sender: TObject);
{ Save the online/offline status of the application to the registry. When
  the user runs the application again, it will launch as it was last
  brought down. }
var
  RegIniFile: TRegIniFile;
begin
  RegIniFile := TRegIniFile.Create(cRegIniFile);
  try
    RegIniFile.WriteBool(cRegSection, cRegOnlineIdent, Online);
  finally
    RegIniFile.Free;
  end;
end;
```

continues

LISTING 34.2 Continued

```
procedure TCustomerDataModule.FilterByCity;
{ If we're online, let the server apply the filter so that we only retrieve
  the records we want. Otherwise, apply the filter to the in-memory result
  set of cdsCust. }
var
  CityStr: String;
  Data: OleVariant;
begin
  InputQuery('Filter on City', 'Enter City: ', CityStr);
  FFilterStr := CityStr;

  if Online then
  begin
    dcomCust.AppServer.FilterByCity(CityStr, Data);
    cdsCust.Refresh;
  end
  else begin
    FFilterType      := ftByCity;
    cdsCust.Filtered := True;
    cdsCust.Refresh;
  end;
end;

procedure TCustomerDataModule.FilterByState;
{ If we're online, let the server apply the filter so that we only retrieve
  the records we want. Otherwise, apply the filter to the in-memory result
  set of cdsCust. }
var
  StateStr: String;
  Data: OleVariant;
begin
  InputQuery('Filter on State', 'Enter State: ', StateStr);
  FFilterStr := StateStr;

  if Online then
  begin
    dcomCust.AppServer.FilterByState(StateStr, Data);
    cdsCust.Refresh;
  end
  else begin
    FFilterType      := ftByState;
    cdsCust.Filtered := True;
    cdsCust.Refresh;
  end;
```

```
end;

procedure TCustomerDataModule.NoFilter;
{ If we're online, let the server apply the filter so that we only retrieve
  the records we want. Otherwise, apply the filter to the in-memory result
  set of cdsCust. }
var
  Data: OleVariant;
begin

  if Online then
  begin
    dcomCust.AppServer.NoFilter(Data);
    cdsCust.Refresh;
  end
  else begin
    FFilterType      := ftNone;
    cdsCust.Filtered := False;
    cdsCust.Refresh;
  end;
end;

procedure TCustomerDataModule.cdsCustFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  case FFilterType of
    ftByCity:  Accept := DataSet.FieldByName('CITY').AsString = FFilterStr;
    ftByState: Accept := DataSet.FieldByName('STATE').AsString = FFilterStr;
    ftNone:  Accept := True;
  end;
end;

end.
```

Initial Wiring

Most of the methods you see in Listing 34.2 are simple methods that perform navigation or manipulation on the client dataset, `cdsCust`. Notice that we provide a method on the data module to expose an operation on `cdsCust` rather than to allow any forms to access it directly. Here, we are just adhering to strict OOP methodologies. Although this is not necessary in Delphi, we do so for consistency and to enforce centralization of database logic.

`CustomerDataModule` contains a `TDCOMConnection` object, `dcomCust`, and the `TClientDataSet` object, `cdsCust`. `DcomCust` is connected to the server application through its `ServerName` property, which is set to `CustServ.CustomerRemoteDataModule`.

CdsCust is linked to qryCust on the remote data module in its ProviderName property. That takes care of the wiring necessary to get a MIDAS application's server and client up and running. However, to get the most out of this technology, some code needs to be written.

Error Reconciliation

After the client application passes changes back to the server, errors may occur (especially in the briefcase model, where it is possible that another user has modified a given record). The error can be handled on the server or on the client. If it is handled on the client, the server passes error information back to the client through the OnReconcileError handler of TClientDataSet. In this event handler, several options are available that we will discuss momentarily. The OnReconcileError property refers to a TReconcileErrorEvent method, which is defined as follows:

```
TReconcileErrorEvent = procedure(DataSet: TClientDataSet; E:
➥EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction)
➥of object;
```

The DataSet parameter refers to the data set on which the error occurred. EReconcileError is an exception class for client dataset errors. You can use this class as you would any exception class. UpdateKind can be any of the values specified in Table 34.1. This information comes from Delphi's online help.

TABLE 34.1 The TUpdateKind Values

TUpdateKind *Value*	*Meaning*
ukModify	The cached update to the record is a modification to the record's contents.
ukInsert	The cached update is the insertion of a new record.
ukDelete	The cached update is the deletion of a record.

The Action parameter is of type TReconcileAction. By setting the Action parameter to raRefresh, the client application cancels any changes made by the user and refreshes its copy of the result so that it is the same as the server's copy. This is what is done in the example. Other options for the Action property may be set to those values shown in Table 34.2, which comes from Delphi's online help (where you can also look for further information on error reconciliation).

TABLE 34.2 The TReconcileAction Values

TReconcileAction *Value*	*Meaning*
raSkip	Skips updating the record that raised the error condition and leaves the unapplied changes in the change log
raAbort	Aborts the entire reconcile operation

`TReconcileAction` *Value*	*Meaning*
`raMerge`	Merges the updated record with the record on the server
`raCorrect`	Replaces the current updated record with the value of the record in the event handler
`raCancel`	Backs out all changes for this record, reverting to the original field values
`raRefresh`	Backs out all changes for this record, replacing it with the current values from the server

Within the `OnReconcileError` handler, you can refer to the `OldValue`, `NewValue`, and `CurValue` properties for each field of the client data set. These are discussed in Chapter 32, "MIDAS Development."

The `OnReconcileError` event handler for `cdsCust`, `cdsCustReconcileError()`, takes care of retrieving the new, old, and current values of any field for which an error has been passed back to the client upon an update. It then invokes the method referred to by `FOnAddErrorToClient`. `FOnAddErrorToClient` is a method pointer of type `TAddErrorToClientEvent` that is defined at the top of Listing 34.2. You will see in our discussion of the application's main form, `MainCustForm`, how we implement a `TAddErrorToClientEvent` method and assign it to `FOnAddErrorToClient`. Again, this is another example of how we try to keep the data module independent of the user interface elements.

Online and Offline Data Manipulation

We have provided a Boolean property, `Online`, whose reader and writer methods take care of putting the client in either its online or offline state. The method that does this is `SetOnline()`.

`SetOnline()` sets `dcomCust.Connected` to `True` if the user is going online (that is, connecting to the server). If the user was previously offline, any pending changes are applied to the server database. Errors will result in the `cdsCust.OnReconcileError` event handler that is executed. If the user is going offline, `dcomCust.Connected` is set to `False`. `CdsCust` will still work with its in-memory copy of the data. In fact, because a filename is specified in `cdsCust.FileName`, the data can be stored locally to a flat file.

`GetOnline()` just returns `True` if the user is online.

NOTE

`TClientDataSet.FileName` is specific to Delphi 4 and 5. If you are running Delphi 3, you can accomplish the same thing by invoking the `SaveToFile()` and `LoadFromFile()` methods of `TClientDataSet`.

Online and Offline Persistence

The `OnCreate` and `OnDestroy` event handlers for `CustomerDataModule` ensure that the client application is run in the same mode (online or offline) as when it was last closed. This is done by storing its state in the System Registry, which is checked every time the application runs. The constants defined at the top of Listing 34.2 specify the Registry section and keys.

Filtering Records

`CustomerDataModule` allows the user to filter out certain records based on the client's city or state of residence. Client-side filtering occurs when the status of the application is offline. When the client is online, the server is allowed to perform the filtering. One thing to note is that when the client issues a filter while online, when he or she goes offline, only those records that were part of the filter are saved locally to the client's machine.

The `FilterByCity()` and `FilterByState()` methods call the `FilterbyCity()` and `FilterbyState()` methods of the application server discussed earlier. These methods are called only if the user is online. If the user is offline, filtering is done via the `Filter` property and the `OnFilterRecord` event handler of `TClientDataSet`.

Client Main Form

The main form for the client application is very straightforward. It is shown in Listing 34.3.

LISTING 34.3 `MainCustFrm.pas—TMainCustForm`

```
unit MainCustFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBNAVSTATFRM, Db, StdCtrls, DBCtrls, Mask, ComCtrls, Menus, ImgList,
  ToolWin, DBMODEFRM, Grids, DBGrids;

type

  TMainCustForm = class(TDBNavStatForm)
    pcClients: TPageControl;
    dsClientDetail: TTabSheet;
    lblFirstName: TLabel;
    lblLastName: TLabel;
    lblCreditLine: TLabel;
    lblWorkAddress: TLabel;
    lblAltAddress: TLabel;
    lblCity: TLabel;
    lblState: TLabel;
    lblZipCode: TLabel;
    lblWorkPhone: TLabel;
    lblAltPhone: TLabel;
    lblCompany: TLabel;
    dbeFirstName: TDBEdit;
    dbeLastName: TDBEdit;
    dbeCreditLine: TDBEdit;
    dbeWorkAddress: TDBEdit;
    dbeAltAddress: TDBEdit;
    dbeCity: TDBEdit;
    dbeState: TDBEdit;
    dbeZipCode: TDBEdit;
    dbeWorkPhone: TDBEdit;
    dbeAltPhone: TDBEdit;
    dbeCompany: TDBEdit;
    tsComments: TTabSheet;
    dbmComments: TDBMemo;
    dsClients: TDataSource;
    SaveDialog1: TSaveDialog;
    OpenDialog1: TOpenDialog;
    mmiSave: TMenuItem;
    N3: TMenuItem;
    mmiApplyUpdates: TMenuItem;
    mmiCancelUpdates: TMenuItem;
    mmiMode: TMenuItem;
    mmiOffline: TMenuItem;
    mmiOnline: TMenuItem;
    tsErrors: TTabSheet;
    lvClient: TListView;
    mmiExit: TMenuItem;
    mmiFilter: TMenuItem;
    mmiByState: TMenuItem;
    mmiByCity: TMenuItem;
    mmiNoFilter: TMenuItem;
    tsClientList: TTabSheet;
    DBGrid1: TDBGrid;
    procedure sbAcceptClick(Sender: TObject);
    procedure sbCancelClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
    procedure sbDeleteClick(Sender: TObject);
    procedure sbFirstClick(Sender: TObject);
    procedure sbPrevClick(Sender: TObject);
    procedure sbNextClick(Sender: TObject);
    procedure sbLastClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
```

continues

LISTING 34.3 Continued

```
    procedure mmiOnlineClick(Sender: TObject);
    procedure mmiApplyUpdatesClick(Sender: TObject);
    procedure mmiCancelUpdatesClick(Sender: TObject);
    procedure dsClientsDataChange(Sender: TObject; Field: TField);
    procedure Exit1Click(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiByStateClick(Sender: TObject);
    procedure mmiByCityClick(Sender: TObject);
    procedure mmiNoFilterClick(Sender: TObject);
  private
    procedure SetControls;
    procedure GoToOnlineMode;
    procedure GoToOfflineMode;

  public
    procedure AddErrorToClient(const aFieldName, aOldValue, aNewValue,
      aCurValue, aErrorStr: String);
  end;

var
  MainCustForm: TMainCustForm;

implementation

uses CustDM;

{$R *.DFM}

procedure TMainCustForm.AddErrorToClient(const aFieldName, aOldValue,
     aNewValue,
  aCurValue, aErrorStr: String);
{ This method is used to add a TListItem to the TListView, aLV. The items
  added here give an indication of the errors that occur when performing
  updates to the server data. }
var
  NewItem: TListItem;
begin
  NewItem := lvClient.Items.Add;
  NewItem.Caption := aFieldName;
  NewItem.SubItems.Add(aOldValue);
  NewItem.SubItems.Add(aNewValue);
  NewItem.SubItems.Add(aCurValue);
  NewItem.SubItems.Add(aErrorStr);
end;
```

```
procedure TMainCustForm.SetControls;
begin
  // Ensure that the navigational buttons are set according to the form's mode.
  sbFirst.Enabled := not CustomerDataModule.IsBof;
  sbLast.Enabled  := not CustomerDataModule.IsEof;
  sbPrev.Enabled  := not CustomerDataModule.IsBof;
  sbNext.Enabled  := not CustomerDataModule.IsEof;

  // synchronize the navigational menu items with the speedbuttons.
  mmiFirst.Enabled    := sbFirst.Enabled;
  mmiLast.Enabled     := sbLast.Enabled;
  mmiPrevious.Enabled := sbPrev.Enabled;
  mmiNext.Enabled     := sbNext.Enabled;

  // Set other menus accordingly

  mmiApplyUpdates.Enabled  := mmiOnline.Checked and (FormMode = fmBrowse) and
    (CustomerDataModule.ChangeCount > 0);
  mmiCancelUpdates.Enabled := mmiOnline.Checked and (FormMode = fmBrowse) and
    (CustomerDataModule.ChangeCount > 0);


  mmiOnline.Checked  := CustomerDataModule.Online;
  mmiOffline.Checked := not mmiOnline.Checked;

  stbStatusBar.Panels[0].Text := Format('Changed Records: %d',
    [CustomerDataModule.ChangeCount]);

  if CustomerDataModule.Online then
    stbStatusBar.Panels[2].Text := 'Working Online'
  else
    stbStatusBar.Panels[2].Text := 'Working Offline'


end;

procedure TMainCustForm.sbAcceptClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.SaveClient;
  SetControls;
end;

procedure TMainCustForm.sbCancelClick(Sender: TObject);
begin
  inherited;
```

continues

LISTING 34.3 Continued

```
  CustomerDataModule.CancelClient;
  SetControls;
end;

procedure TMainCustForm.sbInsertClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.AddClient;
  SetControls;
end;

procedure TMainCustForm.sbEditClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.EditClient;
  SetControls;
end;

procedure TMainCustForm.sbDeleteClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.DeleteClient;
  SetControls;
end;

procedure TMainCustForm.sbFirstClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.First;
  SetControls;
end;

procedure TMainCustForm.sbPrevClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.Previous;
  SetControls;
end;

procedure TMainCustForm.sbNextClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.Next;
  SetControls;
end;
```

```
procedure TMainCustForm.sbLastClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.Last;
  SetControls;
end;

procedure TMainCustForm.FormCreate(Sender: TObject);
begin
  inherited;
  CustomerDataModule.OnAddErrorToClient := AddErrorToClient;
  SetControls;

  // Make these guys refer to each other so that they reset the other
  mmiOnline.Tag := Longint(mmiOffline);
  mmiOffline.Tag := Longint(mmiOnline);
end;

procedure TMainCustForm.GoToOnlineMode;
begin
  CustomerDataModule.Online := True;
  SetControls;
end;

procedure TMainCustForm.GoToOfflineMode;
begin
  CustomerDataModule.Online := False;
  SetControls;
end;

procedure TMainCustForm.mmiOnlineClick(Sender: TObject);
var
  mi: TMenuItem;
begin
  inherited;
  mi := Sender as TMenuItem;

  if not mi.Checked then
  begin

    mi.Checked := not mi.Checked;
    TMenuItem(mi.Tag).Checked := not mi.Checked;

    if mi = mmiOnline then
    begin
      if mi.Checked then
```

continues

LISTING 34.3 Continued

```
        GoToOnlineMode
      else
        GoToOffLineMode
    end

    else begin
      if mi.Checked then
        GoToOfflineMode
      else
        GoToOnlineMode
    end;
  end;
end;

procedure TMainCustForm.mmiApplyUpdatesClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.ApplyUpdates;
  SetControls;
end;

procedure TMainCustForm.mmiCancelUpdatesClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.CancelUpdates;
  SetControls;
end;

procedure TMainCustForm.dsClientsDataChange(Sender: TObject;
  Field: TField);
begin
  inherited;
  SetControls;
end;

procedure TMainCustForm.Exit1Click(Sender: TObject);
begin
  inherited;
  Close;
end;

procedure TMainCustForm.mmiExitClick(Sender: TObject);
begin
  inherited;
  Close;
end;

procedure TMainCustForm.mmiByStateClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.FilterByState;
end;

procedure TMainCustForm.mmiByCityClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.FilterByCity;
end;

procedure TMainCustForm.mmiNoFilterClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.NoFilter;
end;

end.
```

Most of the methods for `TMainCustForm` call the methods of `CustomerDataModule`.

Notice the `AddErrorToClient()` method. This method serves as the `OnAddErrorToClient` property of `CustomerDataModule`. The `OnCreate` event handler of `TMainCustForm` assigns this method to the data module's property. `AddErrorToClient()` adds any events to the `TListView` control on the main form for the user to examine. This `TListView` control displays the field name, old value, new value, and current values for the error. It also displays the error string.

The simple `SetControls()` method handles setting up various controls on the form. It ensures that controls are enabled or disabled when appropriate. The rest of the methods are discussed in the commentary in the source code.

Summary

Although the Client Tracker is a simple application, most of the wiring necessary for creating three-tier applications is shown in this example. You might also find that you need to implement some other specifics such as callbacks or connection pooling, as discussed in Chapter 32, "MIDAS Development." The point is this: Developing three-tier applications using MIDAS is not harder than developing two-tier or even desktop database applications.