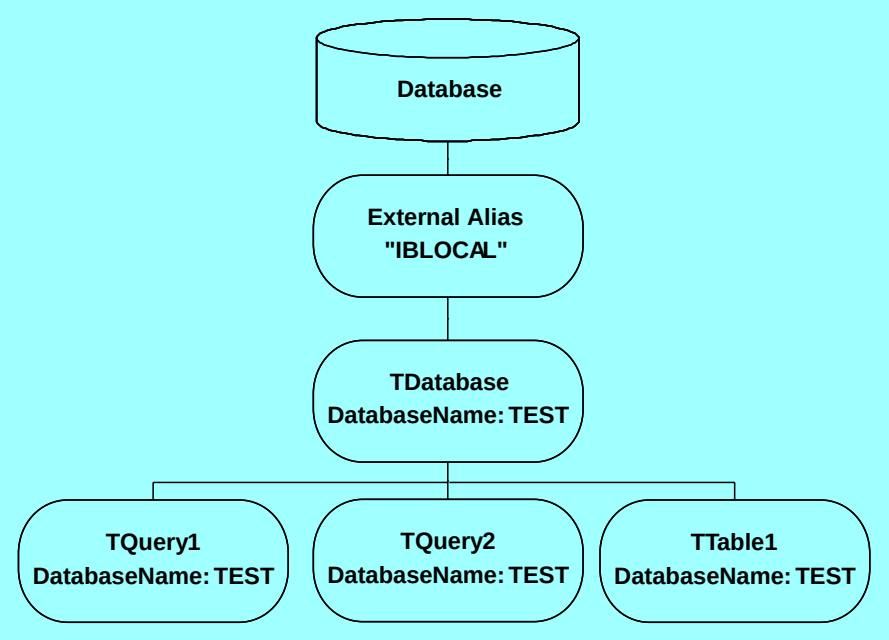
Surviving Client/Server: All About Aliases

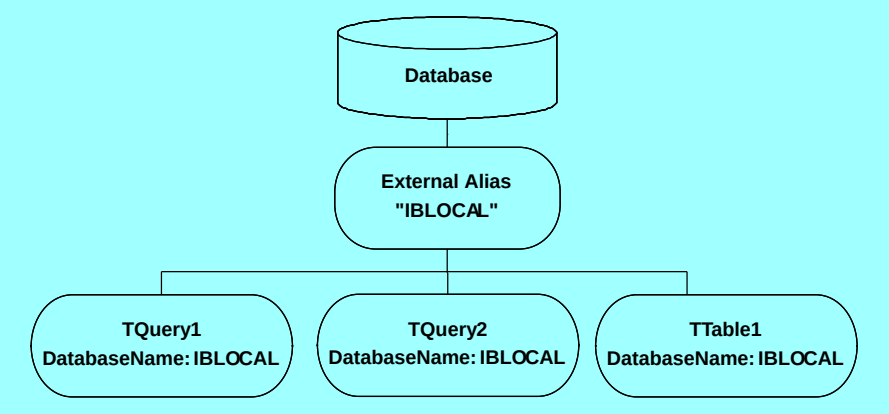
by Steve Troxell

When using the Borland Database Engine (BDE) with Delphi, the most common approach for connecting your application to the database is through an *alias*. An alias is an external definition for the location and type of the database and is set up using the BDE Configuration Utility. Your applications access the database by referring to the alias name. This technique provides a layer of encapsulation that allows you to easily point your application to different physical databases simply by redefining the alias definition.

For example, during development your database will reside in a certain location (file server, hard drive, directory). However, when you deploy the application at a client's site, the location may not be exactly the same as it was back at the office. All you need to do is change the location of the database in the BDE alias at the client's site. No alterations to the program code are needed.



➤ *Figure 1*



➤ *Figure 2*

The TDatabase Component

Within a Delphi program, you typically make the connection between the application and BDE alias through a single `TDatabase` component on the application's main form. Although an explicit `TDatabase` component is not strictly required for a database application, it greatly simplifies managing the database connections. With a `TDatabase` you will centralize the assignment of the external database to the internal dataset components (`TTable`, `TQuery` and `TStoredProc`). By setting the `AliasName` property of the `TDatabase` to the BDE alias for your database, you establish a link between the application and the database. You then assign an *application specific alias* in the `TDatabase`'s `DatabaseName` property. This is a new name you make up for the database that you will use internally within the application for all the dataset components. It is known only within your application, other BDE applications have no way of connecting with this internal alias name.

To complete the chain, you now set the `DatabaseName` property of all your dataset to be the same as the `DatabaseName` of your `TDatabase` component. All your dataset components are now channeled through the centralized `TDatabase` whenever they access the database. This scheme is shown in Figure 1.

Note that it is possible to connect dataset components directly to an external alias by setting the component's `DatabaseName` property to the external alias name. This is illustrated in Figure 2. As you can see from these two illustrations, one advantage of using the `TDatabase` to centralize database

access is that you can instantly change the database reference of all the components in your application by assigning a new AliasName to the TDatabase component.

This may be desirable during development if you kept multiple versions of the database for different purposes. You may have a baseline database used for testing and/or demonstrations and representing the last stable build of the database structure and contents. You may have a separate development database to try out new data modification logic without risking the baseline database. Individual developers may also have local copies of the database. Or, you may be supporting more than one database server and have multiple copies of the database on different platforms.

For development, each of these could have a separate BDE alias and you could easily switch between them by simply changing the AliasName property of the TDatabase component. It is far easier to do this than to keep redefining the same external BDE alias for each database you want to use. It would even be possible to have the alias name assigned to the TDatabase component at runtime through a registry setting or INI file value (note that you should still have a 'default' alias assigned to the TDatabase at design time or you won't be able to see live data at design time).

Controlling The Login Dialog

The application will attempt to log into the database whenever the TDatabase component's Connected property is set to true. This can happen when the Connected property is explicitly set, or when any of the dataset components bound to the TDatabase are activated. The TDatabase component has a default login dialog box which allows the user to enter a username and password. Most developers want to override this spartan dialog box and substitute their own. This is accomplished by adding an OnLogin event handler and setting the Username and Password parameters for the LoginParams list (as shown in Listing 1). Obviously, you will substitute whatever values the user entered in place of the hard coded values shown in this example (a more thorough discussion of this technique can be found in Xavier Pacheco's article *Customized Logins* in the November 1995 issue).

```
procedure TForm1.Database1Login(
  Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['USERNAME'] := 'sysdba';
  LoginParams.Values['PASSWORD'] := 'masterkey';
end;
```

➤ *Listing 1*

```
Database1.StartTransaction;
try
  Query1.ExecSQL;    { INSERT INTO AccountTrans (AcctNo, Amount)
                         VALUES ('123', -100) }
  Query2.ExecSQL;    { INSERT INTO AccountTrans (AcctNo, Amount)
                         VALUES ('789', 100) }
  Database1.Commit;
except
  Database1.Rollback;
  raise;
end;
```

➤ *Listing 2*

Transaction Processing

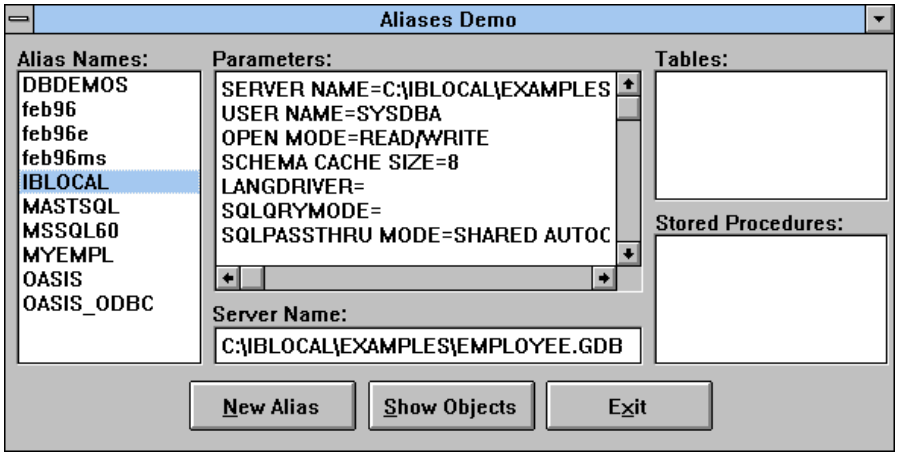
TDatabase is also where you control transaction processing by using the StartTransaction, Commit and Rollback methods. Once you begin a transaction, all subsequent database activity is handled as a single unit; either it all gets written to the database or none of it does. For example, if you needed to transfer funds from one bank account to another, you will have to debit one account and credit the other. But if for some reason one of the operations fails, you want to avoid posting the other: it's all or none. For example, let's say you wanted to transfer $100 from account 123 to account 789. Assume we have two TQuery components (one to debit account 123 and one to credit account 789) bound to a common TDatabase as shown in Listing 2.

Now if either Query1 or Query2 raises an exception, whatever effect either of them had on the database will be 'undone' by the Rollback method. If they are both successful, then execution falls to the Commit method and the changes of both queries are posted in the database.

The TSession Component

When you have dataset components bound to a TDatabase, they are 'aware' of the TDatabase through the Session variable. Session is an instance of TSession that is automatically created for you (just as Application is an automatically created instance of TApplication). Session manages all of the database components within an application and, as we'll see in a moment, provides a wealth of functionality to examine BDE aliases.

Since the dataset components can communicate with the TDatabase component through Session, you are free to create forms and units that may be shared among more than one application. Suppose, for example, you have an integrated system of multiple applications all accessing the same database. Shared forms or units could be compiled into different applications as long as the dataset components use the same DatabaseName as the application's TDatabase. This can be achieved by adopting a standard naming convention for DatabaseName properties within the applications, or by passing the DatabaseName into the unit and having the unit assign it to the dataset components programmatically.



➤ *Figure 3*

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Session.GetAliasNames(AliasNames.Items);
  AliasNames.ItemIndex := 0;
  AliasNamesClick(nil);
end;
procedure TMainForm.AliasNamesClick(Sender: TObject);
begin
  AliasTables.Clear;
  AliasProcs.Clear;
  with AliasNames do
    Session.GetAliasParams(Items[ItemIndex], AliasParams.Lines);
  AliasServerName.Text := AliasParams.Lines.Values['SERVER NAME'];
end;
procedure TMainForm.ShowDBObjectsBtnClick(Sender: TObject);
var
  I: Integer;
begin
  with AliasNames do begin
    Session.GetTableNames(Items[ItemIndex], '',
       False, False, AliasTables.Items);
    try
      Session.GetStoredProcNames(Items[ItemIndex], AliasProcs.Items);
    except
    end;
  end;
end;
```

➤ *Listing 3*

You should note that an application's `Session` variable is not available to DLLs loaded by that application. Nor can you pass database VCL components as parameters into a DLL routine. The database VCL is heavily bound to the application's BDE environment and DLL usage of the application's database components could crash the system.

One technique for encapsulating dataset components within a DLL is to pass the BDE alias name used by the application and the username and password used to log in, and have the DLL dynamically create an instance of `TDatabase` using those values to make an independent connection to the database. The DLL's dataset components could then use this database connection. This does result in an extraneous login to the database so you'll have to judge for yourself whether or not to use this approach.

Reading IDAPI.CFG

Information for all the external BDE aliases are stored in the file IDAPI.CFG. You can examine this file from within a Delphi application through the `Session` variable. The program shown in Figure 3 and Listing 3 illustrates how this is done (this example program is available on this issue's free disk).

When the program starts up, we use `TSession`'s `GetAliasNames` method to return a list of all aliases defined in IDAPI.CFG. Then we use the method `GetAliasParams` to display the parameters for the currently selected alias. You'll notice that the parameter list returned by `GetAliasParams` is conveniently of the form `<param>=<value>`, so we can use the `Values` property in the `TStrings` class to look up any particular parameter we want. In this example, we are showing the server name parameter for the currently selected alias in the edit box in the middle of the form.

```
procedure GetTableNames(
  const DatabaseName,
  Pattern: string;
  Extensions, SystemTables:
  Boolean; List: TStrings);
```

➤ *Listing 4*

Finally, if the user clicks the `Show Objects` button, we can use the `Session` variable to obtain a list of all the tables and all the stored procedures (if applicable) in the database pointed to by the alias. We use `GetTableNames` to see all the tables and `GetStoredProcNames` to see all the stored procedures. This information is not stored within IDAPI.CFG, so when you use either of these two methods, Delphi attempts to log into the database to obtain these lists.

The declaration for procedure `GetTableNames` is shown in Listing 4. Several additional parameters are provided for filtering the list of table names. `DatabaseName` is really the alias name you are interested in. With the `Pattern` parameter, you can supply a wildcard match string to filter the table names. Either the standard DOS tokens (`*` and `?`) or the SQL tokens (`%` and `_`) can be used. For desktop databases, when the `Extensions` parameter is true the filename extensions are included in the table names returned. For SQL databases, when the `SystemTables` parameter is true internal system tables are also included in the return list.

You can add a new alias by clicking the `New Alias` button. This launches the dialog shown in Figure 4 with corresponding code shown in Listing 5. We will be using

the BDE API function DbiAddAlias to write the new alias into IDAPI.CFG. This form gathers the information needed by DbiAddAlias.

First we use Session's method GetDriverNames to fill the DriverType combo box dropdown list with all the currently installed BDE drivers. The user enters a name for the new alias. Then, when the user selects a driver from the dropdown list, we use Session's method GetDriverParams to fill the grid with the appropriate parameters for that BDE driver along with the default values for the driver.

To do this we loop through the list of driver parameters and split each string apart into the parameter name and parameter value and place each piece in one of the grid columns. We use a simple utility routine called Split to break the string up (the code is not shown here, but can be found in the example project on this issue's free disk). The user now has the opportunity to change any or all of the parameters shown before clicking OK to save the new alias.

Ironically, although the PASSWORD parameter is actually returned by GetDriverParams, DbiAddAlias does not accept it as a valid parameter for SQL drivers. So, we must deliberately exclude this from the list of parameters.

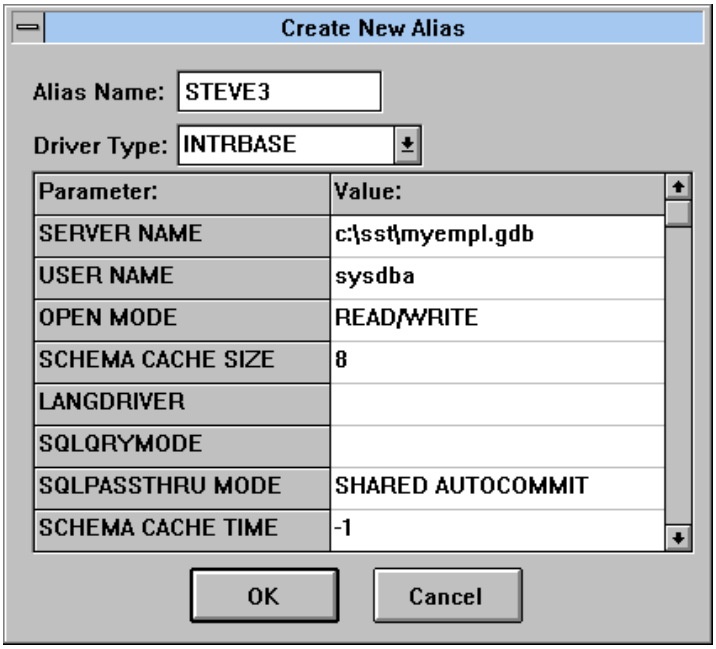
When saving the new alias, the program ultimately falls into the SaveAlias routine at the bottom of the listing. Here we copy all the driver parameters into a single null-terminated string in the format of <parametername>: <value> with a semi-colon between each parameter (be sure not to include a semi-colon after the last parameter). To actually write the new alias in the IDAPI.CFG file, we use DbiAddAlias. This function is defined in the DbiProcs unit and the interface definition for this unit can be found in file \DELPHI\DOC\DBIPROCS.INT. The declaration for this function is shown in Listing 6.

Cfg is a handle to the BDE configuration for the current session. This is not needed for BDE version 2.5 so we simply pass nil. AliasName and DriverType simply contain the names of the alias and driver

```
procedure TNewAliasForm.FormCreate(Sender: TObject);
begin
  Session.GetDriverNames(Driver.Items);
  with DriverParamsGrid do begin
    Cells[0, 0] := 'Parameter:';
    Cells[1, 0] := 'Value:';
    RowCount := 2;
  end;
end;
procedure TNewAliasForm.DriverClick(Sender: TObject);
var DriverParamsList: TStringList;
    I: Integer;
    ParamName: String;
    ParamValue: String;
begin
  DriverParamsList := TStringList.Create;
  try
    with Driver do
      Session.GetDriverParams(Items[ItemIndex], DriverParamsList);
    { Populate the driver parameters grid }
    with DriverParamsGrid do begin
      RowCount := 1;
      for I := 0 to DriverParamsList.Count - 1 do begin
        Split(DriverParamsList.Strings[I], '=', ParamName, ParamValue);
        if ParamName <> 'PASSWORD' then begin
          Cells[0, I + 1] := ParamName;
          Cells[1, I + 1] := ParamValue;
          RowCount := RowCount + 1;
        end;
      end;
      FixedRows := 1;
      SetFocus;
    end;
  finally
    DriverParamsList.Free;
  end;
end;
procedure TNewAliasForm.OkBtnClick(Sender: TObject);
begin
  try
    { Evil things happen to BDE when the alias is an empty string }
    if Alias.Text = '' then
      raise Exception.Create('Must supply an alias name');
    if Driver.ItemIndex = -1 then
      raise Exception.Create('Must supply a driver type');
    SaveAlias;
  except
    ModalResult := mrNone;
    raise;
  end;
end;
procedure TNewAliasForm.SaveAlias;
var AliasName: array[0..25] of char;
    DriverName: array[0..25] of char;
    Params: PChar;
    TempStr: array[0..255] of char;
    I: Integer;
begin
  StrPCopy(AliasName, Alias.Text);
  StrPCopy(DriverName, Driver.Text);
  Params := StrAlloc((DriverParamsGrid.RowCount - 1) * 255);
  try
    { Assemble the driver parameters in a single null-terminated string }
    StrPCopy(Params, '');
    with DriverParamsGrid do begin
      for I := 0 to RowCount - 2 do begin
        StrPCopy(TempStr, Cells[0, I + 1] + ': ' + Cells[1, I + 1] + ';');
        StrCat(Params, TempStr);
      end;
    end;
    case DbiAddAlias(nil, @AliasName, @DriverName, Params, True) of
      DBIERR_INVALIDPARAM  : raise Exception.Create('Invalid Param');
      DBIERR_NAMENOTUNIQUE : raise Exception.Create('Alias already exists');
      DBIERR_OBJNOTFOUND   : raise Exception.Create(
                                       'Invalid driver parameter');
      DBIERR_UNKNOWNDRIVER : raise Exception.Create('Invalid driver');
    end;
  finally
    StrDispose(Params);
  end;
end;
```

➤ *Listing 5*

➤ *Figure 4*



respectively. `Params` is the string of driver parameters we assembled from the grid. This list need not contain all of the driver parameters. The default values defined within the BDE are assumed for any missing parameters. If `Params` is nil, then default values are assumed for all driver parameters. `Persist` simply indicates if this is a permanent alias (if true) written to IDAPI.CFG, or an application-specific alias (if false) for the current session only. The error values returned by `DbiAddAlias` are defined in the `DbiErrs` unit (whose interface section can be found in \DELPHI\DOC\DBIERRS.INT).

Summary

We have seen how Delphi helps us to manage database connections through the `TDatabase` component and how we can obtain a great deal of information about all the available aliases through the `TSession` class. In addition, we have also seen how to create our own aliases through code using the BDE API. Delphi provides more database functionality than one might think, if you know where to look.

```
function DbiAddAlias(
  Cfg: Pointer;
  AliasName, DriverType, Params:
  PChar; Persist: Bool): Word;
```

➤ *Listing 6*

Steve Troxell is a Software Engineer with TurboPower Software where he is developing Delphi Client/Server applications using InterBase and Microsoft SQL Server for parent company Casino Data Systems. Steve can be reached on the internet at stevet@tpower.com and also on CompuServe at 74071,2207