Surviving Client/Server: Managing User Logins, Part 3

by Steve Troxell

This month, we'll make our final improvements to the TLoginManager component we've been building. As before, we'll be revising code we first built in previous issues. To save space, I'll only show partial code listings to point out changes made to previous code.

The complete code for the TLoginManager system can be found on this month's companion disk in the SURVIVE directory.

Server/Database Overrides

The first new feature we want to implement is a runtime override for the server name and/or database name defined in the alias. Normally Delphi applications know which physical database to connect to through a definition within an external BDE alias. The alias name is hardcoded within the application's TDatabase.AliasName property. If we want to connect to a different database, we change the alias definition through the BDE Configuration Utility.

However, sometimes even this level of flexibility is not enough. You may have several versions of your database on one server: most recent stable build available for demos, a work-in-progress build, a testing build, individual versions for each developer, etc. Or, you may have different physical servers such as a development server at one location and the demonstration and testing server at another. As you distribute copies of the application around to these various sites, it is out of the question to keep changing the hard-coded alias name within the application and cumbersome to say the least to expect people to keep redefining the same alias within the BDE Config Utility.

Our solution is to keep a static BDE alias definition and introduce "overrides" for the server and database whenever they differ from the "defaults" defined by the application. One way to implement this would be to use an INI file to define the override values. If no INI file information is found, then the default values from the application are used as normal. Another approach would be to have the user enter or select a server/database name when they attempt to login, with one option being the "default" database.

Database parameter values in the TDatabase.Params property take precedence over values for the same parameters defined in the external BDE alias. Therefore to implement a runtime override for either the server or the database to connect to, we simply assign the correct parameter values through TDatabase.Params before we connect the database component.

We'll add the SetServerName and SetDatabaseName methods to TLoginManager which will allow us to set these parameters should we detect overrides at runtime. In addition, if we set one of these parameters to an empty string value, we will take that to mean "restore to the original value provided by the application". We'll also provide two read-only properties to show the current values for each of these parameters. The additional code we need in LOGIN.PAS is shown in Listing 1 (over the page).

Since the override values must be set in the application's TDatabase component, and because there are various ways to obtain the override values (INI file, dialog, etc), we won't make any assumptions about which comes first, setting the override values or setting the MainDB property. We will write our code safely enough such that neither is dependent on the other occurring first.

The first thing we must take into account is that the "default" value for the server or database name could come from one of two sources. Obviously the BDE alias is one place. But less obviously, the calling application could have defined its own values within its TDatabase component (independent of TLoginManager). Since we will be writing our override values within the TDatabase's parameters, we'll destroy any value that might already be there. In order to provide the ability to restore to default values, we'll need to capture the existing values when the TDatabase component is registered through the MainDB property.

The first thing we need is the helper method GetDBParamValue. We'll use this to obtain the existing values from the application. When given a database parameter name, this method returns the corresponding value defined in the application's TDatabase component. Failing that, it extracts the value from the BDE alias definition by reading the Session variable.

Next, we'll modify the SetMainDB method to capture the existing name values. If no name overrides have been defined yet, these same values are used as the return values for the ServerName and DatabaseName properties.

If name overrides have already been defined, we set them in the TDatabase component using the SetDBParamValue helper method. Remember last month we added an internal TDatabase component to handle certain tasks. So whenever we change database values, we must remember to change both the application's TDatabase component as well as our own internal one.

Name overrides are registered using the SetDatabaseName and SetServerName methods. If MainDB has already been set, we go ahead and

write these values into the TDatabase parameters. If we are setting the value to an empty string, then we write the original default value in the TDatabase parameters.

Now that the building blocks are in place, just exactly how do we get the database overrides from the application into TLoginManager? There are a few different approaches depending on how you want to define the overrides.

One possible technique is to define overrides in an INI file that resides with the EXE file. If the INI filenames and contents are standardized across your product line, you may want to embed the code to read the INI file and call SetServerName and SetDatabaseName in the TLoginManager.Create method. This has the advantage of further centralizing the business rules regarding logins, but the disadvantage that the override names cannot be changed without a restart of the application.

Another approach is to make TLoginManager's ServerName and DatabaseName properties read/write (by binding the SetServerName and SetDatabaseName methods as write methods), have the application read the INI files, then set the names as part of the TLoginManager registration process.

The technique I have decided to show you in this article is to add server name and database name arguments to the TLoginManager.Login method. This allows us to provide edit controls or combo boxes in the login dialog itself so the user may select the server and database they want each time they login. The INI file approach still works too. The application reads the INI file, stores the values and passes them into the Login method whenever it is called. This approach seems to provide the most flexibility for various methods of defining the override names.

Listing 2 shows the changes we need to make to the Login method. Note that a TDatabase component must be disconnected before we can change its parameters. So we must set the override values after we've logged out the previous user.

Remember that we've built these methods such that if an empty string is passed in, the default names defined by the application or the alias are used. So, if an application does not support overrides, or if we want it to reset to the default values, we simply have to pass in empty strings for the Server and Database parameters.

➤ *Listing 1*

```
TLoginManager = class(TComponent)
  protected
    ...
    FCallersDatabaseName: string;
    FCallersServerName: string;
    FDatabaseName: string
    FServerName: string
    function GetDBParamValue(ParamName: string): string;
    procedure SetDBParamValue(ParamName, Value: string);
    procedure SetDatabaseName(Value: string);
    procedure SetServerName(Value: string);
  public
    ...
    property DatabaseName: string read FDatabaseName;
    property ServerName: string read FServerName;
end;

implementation

function TLoginManager.GetDBParamValue(
  ParamName: string): string;
{ Returns the value for the given database parameter. }
var DBParams: TStringList;
begin
  { First, check for specific values in the
    application's main TDatabase component. }
  Result := FMainDB.Params.Values[ParamName];
  { Failing that, get value from alias definition. }
  if Result = '' then begin
    DBParams := TStringList.Create;
    try
      Session.GetAliasParams(
        FMainDB.AliasName, DBParams);
      Result := DBParams.Values[ParamName];
    finally
      DBParams.Free;
    end;
  end;
end;

procedure TLoginManager.SetDBParamValue(
  ParamName, Value: string);
begin
  FMainDB.Params.Values[ParamName] := Value;
  LoginDM.dbInternal.Params.Values[ParamName] := Value;
end;

procedure TLoginManager.SetMainDB(Value: TDatabase);
begin
  if Value <> FMainDB then begin
    FMainDB := Value;
    LoginDM.dbInternal.AliasName := FMainDB.AliasName;
    {... Existing code omitted ...}
    { ServerName and/or DatabaseName overrides could
      have been registered already. If not, then we must
      make sure ServerName and DatabaseName properties
      return the values given in the application's
      TDatabase component, or in the alias definition. }
    FCallersServerName := GetDBParamValue('SERVER NAME');
    if FServerName <> '' then
      SetDBParamValue('SERVER NAME', FServerName)
    else
      FServerName := FCallersServerName;
    FCallersDatabaseName :=
      GetDBParamValue('DATABASE NAME');
    if FDatabaseName <> '' then
      SetDBParamValue('DATABASE NAME', FDatabaseName)
    else
      FDatabaseName := FCallersDatabaseName;
  end;
end;

procedure TLoginManager.SetDatabaseName(Value: String);
begin
  FDatabaseName := ANSIUppercase(Value);
  { If MainDB has already been registered... }
  if FMainDB <> nil then begin
    if FMainDB.Connected or
      LoginDM.dbInternal.Connected then
      raise Exception.Create(
        'Cannot set TLoginManager.DatabaseName once '+
        'database is connected');
    if FDatabaseName = '' then
      FDatabaseName := FCallersDatabaseName;
    SetDBParamValue('DATABASE NAME', FDatabaseName);
  end;
end;

procedure TLoginManager.SetServerName(Value: string);
begin
  FServerName := ANSIUppercase(Value);
  { If MainDB has already been registered... }
  if FMainDB <> nil then begin
    if FMainDB.Connected or
      LoginDM.dbInternal.Connected then
      raise Exception.Create(
        'Cannot set TLoginManager.ServerName once '+
        'database is connected');
    if FServerName = '' then
      FServerName := FCallersServerName;
    SetDBParamValue('SERVER NAME', FServerName);
  end;
end;
```

One final chore we must handle is to clean up the password change code to allow us to change the password for the correct server and database if overrides are permitted.

If you recall from last month we implemented the code that changes the password in a DLL. Now we must add two parameters to the exported function to allow us to pass in the optional server name and database name overrides. Inside the DLL function, we then take these values into account when setting up the DLL's database connection (see Listing 3). When the function is called from `TLoginManager`, we simply pass in the values of `FServerName` and `FDatabaseName`.

Handling Multiple Databases

You may have need for one or more of your applications to access more than one physical database at a time. There are a number of issues in handling the interaction between `TLoginManager` and additional databases, most of which are very closely tied to the specific RDBMS platforms involved and exactly how users are set up across them. For our purposes here, we'll only deal with connecting and disconnecting all the databases when the user "logs in". We'll further assume that the user has the same username and password on all the relevant databases. After getting past this, we'll discuss some of the other issues and possible techniques for addressing them.

➤ *Listing 2*

```
procedure TLoginManager.Login(UserName, Password, Server, Database: string);
begin
  Logout;
  FUsername := Username;
  FPassword := Uppercase(Password);
  { Deal with possible server/database name overrides }
  SetServerName(Server);
  SetDatabaseName(Database);
  { Connect to physical database }
  Connect;
  {... Existing code omitted for brevity ...}
end;
```

➤ *Listing 3*

```
{ function interface }
function ChangePassword(iAliasName : PChar; iServerName : PChar;
  iDatabaseName : PChar; iUserName : PChar; iOldPassword : PChar;
  iNewPassword  : PChar; var oErrMsg   : PChar): Word; export;
{ code fragment from function body }
with TempDatabase do begin
  AliasName := StrPas(iAliasName);
  DatabaseName := 'PasswordChangeDB';
  { The following lines were added for server/database overrides }
  if Assigned(iServerName) and (StrPas(iServerName) <> '') then
    Params.Values['SERVER NAME'] := StrPas(iServerName);
  if Assigned(iDatabaseName) and (StrPas(iDatabaseName) <> '') then
    Params.Values['DATABASE NAME'] := StrPas(iDatabaseName);
  Params.Values['USER NAME'] := StrPas(iUserName);
  Params.Values['PASSWORD'] := StrPas(iOldPassword);
  LoginPrompt := False;
  Connected := True;
end;
```

➤ *Listing 4*

```
TDBList = class(TList)
  public
    function Add(Item: TDatabase): Integer;
  end;
function TDBList.Add(Item: TDatabase): Integer;
begin
  Result := inherited Add(Item);
end;
```

The concept we're talking about here is having two or more `TDatabase` components in your application. One of these components points to the "main" database, that is, the database containing our `Users` and `AuditTrail` tables. This is the principal database users are logged into by all applications. The rest are ancillary databases containing application data. Different applications may have different ancillary databases (or none), but all will connect to the "main" database.

Obviously, our main `TDatabase` component is registered with `TLoginManager` through the `MainDB` property, just as we've been doing all along. As for the rest of the `TDatabases`, all we need to do is inform `TLoginManager` of their presence to allow it to control the setting of their `Connected` properties.

We'll use a `TList` field in `TLoginManager` to keep track of all the `TDatabases` used by the program. We'll create a descendant component class called `TDBList` as shown in Listing 4 just so we can type-check exactly what is put into the list.

Next, we'll add a new property to `TLoginManager` as shown in Listing 5. With this done, our calling applications will register their ancillary databases as shown in Listing 6.

In order for anything to happen with these additional databases, we must extend the login and logout code to affect them as well. Listing 7 shows the new Connect and Disconnect methods we need do this.

That's all that is needed to have `TLoginManager` automatically connect and disconnect all registered databases with the same username and password. Obviously what you have `TLoginManager` do with your ancillary databases is very particular to what you want to do with them. Let's discuss some of the possibilities.

The code we have now assumes that the server and database name overrides we talked about at the beginning of this article only apply to the "main" database. That might be valid if, for example, your ancillary databases are existing in-house systems and you would

never have the possibility of overriding them. However, if the multiple databases in your system are inherent to its design and you can assume all of them are on the same server, you would want to modify SetServerName to loop through FApplicationDBs and override their server names as well.

Another possibility is that the user names or passwords might not be the same across all databases. You could extend the Users table (or create an extra table) to contain the corresponding user names and passwords for the other databases as well. TLoginManager.Connect could then use GetDBParamValue to examine the database name for each ancillary database, lookup a new username and password value, and pass them into ConnectDB rather than passing FUsername and FPassword.

What happens when a user changes their password? That depends on what you want to happen. In the case where all user names and passwords are the same across all databases, you'll have to change PASSWORD.DLL to accept a list of databases rather than info on just one database. You'll also have to provide more extensive error handling to cover the possibility that not all of the password changes were successful.

If different user names and passwords are used on different databases, it might be best to just have the "password change" function affect the main database. Changing passwords in other databases would then be a function of those other systems' administration software. Or, you could make more elaborate password change dialogs to allow users to select the database for which they are changing their password and write dedicated code for each database.

If you decide to implement multiple database access, think very carefully about how the different databases interact and adjust the TLoginManager code accordingly.

Conclusion

This is the end of our TLoginManager component. My goal was to show you some techniques for centralizing business rules and logic relevant to any database system. Along the way we added a lot of features that would have been extremely difficult to handle consistently across more than one product had we not centralized the code. By the way, all of the functionality we've discussed has been used in real-world projects developed here at TurboPower.

Next month, we'll look at the other side of the login manager by developing a system administrator application which transparently binds our Users table with the RDBMS. We use this app to add, change, or delete users and both the native RDBMS and our custom extensions are updated accordingly.

Steve Troxell is a Senior Software Engineer with TurboPower Software. He can be reached by email at stevet@tpower.com or on CompuServe at 74071,2207

➤ *Listing 5*

```
TLoginManager = class(TComponent)
  {... Existing code omitted ...}
  protected
    FApplicationDBs: TDBList;
  public
    property ApplicationDBs: TDBList
      read FApplicationDBs write FApplicationDBs;
  end;
constructor TLoginManager.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  {... Existing code omitted ...}
  FApplicationDBs := TDBList.Create;
end;
destructor TLoginManager.Destroy;
begin
  FApplicationDBs.Free;
end;
```

➤ *Listing 6*

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  { Register the LoginManager component }
  with LoginManager do begin
    ApplicationID := 1;
    MainDB := dbAppMain;
    { repeat following line for each ancillary database in the application }
    ApplicationDBs.Add(dbPubs);
    OnLogin           := LoginManagerLogin;
    OnLogout          := LoginManagerLogout;
    OnBadLogin        := LoginManagerBadLogin;
    OnPasswordExpired := LoginManagerPasswordExpired;
  end;
end;
```

➤ *Listing 7*

```
procedure TLoginManager.Connect;
var I: Integer;
begin
  ConnectDB(FMainDB, FUsername, FPassword);
  for I := 0 to FApplicationDBs.Count - 1 do
    ConnectDB(FApplicationDBs.Items[I], FUsername, FPassword);
end;
procedure TLoginManager.Disconnect;
var I: Integer;
begin
  DisconnectDB(FMainDB);
  for I := 0 to FApplicationDBs.Count - 1 do
    DisconnectDB(FApplicationDBs.Items[I]);
end;
```