# *Surviving Client/Server:* Managing User Logins, Part 1

*by Steve Troxell*

Any client/server database application which is sensitive to user security is going to be very concerned about managing the login process and recording certain facts about the currently logged in user. The projects I've been involved with until recently are casino management applications designed to handle monetary transactions and transfer of funds for multi-million dollar casinos. Keeping track of who is doing what to that money is a very big concern. Some features we incorporate into our software products include:

- Requiring users to change their password after a certain number of days (the number of days varying from person to person);
- Posting all logins (including unsuccessful logins) and logouts, among many other things, to an audit trail table in the database;
- Controlling which menu items and modules are visible to the user based on their particular access rights to the system;
- Setting various internal limits and flags based on the user's authorization to perform tasks.

Obviously, we want to make sure all these tasks are handled consistently and accurately by all the software in our product line, so that means reusable code. However, we also want to retain the capability for a particular application to do specific processing related to logging in and out if needed, so that means something like event handlers. Finally, we want to minimize our dependency on any particular back end server, so that means not relying on vendor-supplied user identifiers, password management or audit systems.

## Functional Requirements

We're going to develop a login manager to take care of all the dirty work of user connections. The specific requirements are listed in Figure 1. This manager is a scaled-down version of the system TurboPower's Custom Group actually uses for our casino software – the requirements for your system may be different. The purpose here is to show you a way of handling some login related activities consistently across a product line. From this, you should have plenty of ideas about how to implement your specific needs.

- Connect the TDatabase to the server.
- Post login to the database audit trail (even if unsuccessful).
- Post logout to the database audit trail.
- Determine if the user's password has expired, if so provide change password dialog and reset next date of expiration.
- Allow user to voluntarily change password and reset next date of expiration.
- Post the user's last date and time of login.
- Obtain the user's system ID number (to be associated with all transactions posted during this session).
- Allow runtime override of alias definition for the database to connect to.
- Allow runtime override of alias definition for server to connect to.
- Provide centralized event-handlers for each application to perform specific tasks during login and logout.

➤ *Figure 1: Login Manager requirements*

**For Login:**

- Log out existing user (if any).
- Make connection to database(s) for current user.
- If connection unsuccessful:
  - Allow application-specific processing of unsuccessful login,
  - Post an unsuccessful login to the system audit trail.
- Load database values for current user (full name, date password last changed, etc).
- Check to see if user's password has expired; if so, force them to change password or abort login.
- Allow application-specific login events to occur (these may abort the login if needed).
- Post successful login to the system audit trail.
- Update last login date/time for current user.
- Allow application-specific post-login events to occur (these cannot abort the login).

**For Logout:**

- Allow application-specific logout events to occur (these may abort the logout if needed).
- Break the connection to the database(s) for the current user.
- Post completed logout in the system audit trail.

➤ *Figure 2*

## Functional Design

Our login manager will have three main functions: login, logout and voluntary password change. We're going to save password changing for next month, so for now we'll just be concerned with login and logout. Figure 2 shows the specification of these functions.

The TDatabase component alone is inadequate to accomplish this. Too much code would have to be crammed into the OnLogin event handler that wouldn't be reusable by other applications sharing the same database. The handling of database parameter lists and login parameter lists is cumbersome (and again isn't reusable across applications). There's no event handler for logging out, so that function must be coded independently of the rest of the connectivity functions in TDatabase.

It's obvious that we could make a descendant component from TDatabase and add functionality to cover all these objections. That is certainly an acceptable approach. However, there would still be problems if your system must connect to multiple databases. For example, the company may be migrating from an AS/400 system to an Oracle client/server system and parts of both must be accessible to the front-end.

Our login manager will be implemented as a class called TLoginManager and instantiated as a permanent system variable called LoginManager (just as Delphi's Application variable is a permanent instantiation of the TApplication class). Why not make a standard component that we can drop onto any form from the component palette? Because we're going to use login information such as username, proper name and user ID throughout the system and this data must persist. Each time we drop a component onto a form, we get a local instance of that component – the data isn't persistent between forms.

Any application using LoginManager will still use its own TDatabase component for connecting to the database: all application data access is done normally through TDatabase and other data components. However, the application will not directly connect and disconnect to the database by setting TDatabase.Connected. Instead, the application "registers" itself with LoginManager at program startup by passing pointers to its TDatabase component and event handler routines which will take care of application specific logic. Listing 1 shows typical code in the application's main form to register with LoginManager. MainDB is a TDatabase property of LoginManager and the Onxxxx properties are all event handlers for LoginManager.

```
procedure TMain.FormCreate(Sender: TObject);
begin
  { Register with the login object }
  with LoginManager do begin
    { You should have a global constant in your applications }
    ApplicationID := 4;
    { ApplicationDB is the name of the app's TDatabase component }
    MainDB := ApplicationDB;
    { These are local app procedure names to handle login events }
    OnLoggingIn  := LoginManagerLoggingIn;
    OnLogin      := LoginManagerLogin;
    OnLoggingOut := LoginManagerLoggingOut;
    OnLogout     := LoginManagerLogout;
    OnBadLogin   := LoginManagerBadLogin;
  end;
end;
```

➤ *Listing 1*

Because of our requirement to post logins and logouts to the audit trail, we also want to identify which program in the product line the user was logging in from. To accommodate this, each application is assigned a unique identifier which is passed along to the audit trail. As part of the registration process, we inform LoginManager of the application's identifier through LoginManager.ApplicationID.

When the application needs to login a user, it captures the username and password with a local dialog of some sort and passes those values into LoginManager.Login. This method performs the actual database connection by setting the username and password parameters in the application's TDatabase component and then setting the Connected property to True.

## Database Design

In order to support the requirements listed in Figure 1, we must have some data structures in place to record this information.

For every user we need to keep track of their username, proper name, user ID number, date/time of last login, date/time of last password change and how long their password remains valid before requiring a change. For this purpose, we define a table as shown in Listing 2.

```
CREATE TABLE Users(
  UserID                  int,        /* System ID number */
  Username                char(30),   /* Login user name */
  FirstName               char(20),   /* User's proper first name */
  LastName                char(20),   /* User's proper last name */
  DateLastLogin           datetime,   /* Date and time of last login */
  DateLastPasswordChange  datetime,   /* Date and time of last
                                         password change */
  PasswordLifespan        smallint)   /* Number of days between forced
                                         password changes */
```

➤ *Listing 2*

You may wonder why we're storing the user's login name but not their password. We'll let the RDBMS manage the true username and password for login purposes and we'll use the RDBMS's API to change users' passwords. Our supplementary table of user information includes the username field in order to associate the user logging in with a record in our database.

In addition to user information, we need an audit trail table to record every time a user logs in, logs out, changes their password, etc.

To support the audit trail, we define a table as shown in Listing 3.

Obviously, these tables are simplified for illustration. In reality, the tables you use for your system may contain much more information and may be organized very differently.

## TLoginManager

Now we're ready to start building our login manager. `TLoginManager` is implemented in two units: `LOGIN` is a standard Delphi unit and contains the class definition and implementation of all the class methods. `DMLOGIN` is a data module (called `LoginDM`) and contains all data components and code to support the login/logout processes (for example, queries to post messages in the audit trail, change the last login date of the user, etc). `DMLOGIN` contains only data components relevant to `TLoginManager`, it is not accessible by the application.

Listing 4 shows the interface to the `TLoginManager` class. This includes all the class fields and the basic methods and properties (we'll build these up as we go along). We won't get into the specifics of component construction here, our focus is to understand the database mechanisms.

As you can see, the `LoginManager` global variable is created automatically by the `initialization` block of the unit. Note that we need not free the variable in a `finalization` block, because it is automatically freed when its owner, the `Application` variable, is freed. Also, within `TLoginManager`'s constructor method we are creating the `LoginDM` data module (again, we don't have to explicitly free it). This keeps the data module hidden from the calling application. Otherwise, the application would have to use the data module unit and set it for auto-creation at program startup.

When the application registers its `TDatabase` component with `LoginManager`, the `SetMainDB` routine loops through the `LoginDM` data

➤ *Listing 3*

```
CREATE TABLE AuditTrail(
  AuditTrailID       int identity,  /* Auto-increment key field */
  ApplicationID      smallint,      /* Identifies app originating event */
  EventID            smallint,      /* Identifies the event (login, logout,
                                       change password, etc) */
  Timestamp          datetime,      /* Date and time of the event */
  UserID             int,           /* Identifies user originating event */
  Description        varchar(255))  /* Message associated with event */
```

➤ *Listing 4*

```
unit Login;
interface
uses Classes, Forms, SysUtils, DB, DBTables;
const
  cNoUserID         = -1; { Token for no user connected }
  cNoAppID          = -1; { Token for no application defined}
  DefNumAttempts    = 3;  { Default number of login retry attempts }
type
  TLoginEvent = procedure(Sender: TObject; UserName: string;
     Password: string) of object;
  TLoggingInEvent =
     procedure(Sender: TObject; UserName: string;
     Password: string; var Cancel: Boolean) of object;
  TLoggingOutEvent = procedure(Sender: TObject;
     var Cancel: Boolean) of object;
TLoginManager = class(TComponent)
  protected
    { Identifier for the application }
    FApplicationID: LongInt;
    { Date/time of the last login for this user }
    FDateLastLogin: TDateTime;
    { True if user's password has expired on this login }
    FPasswordExpired: Boolean;
    { Pointer to the application's TDatabase component }
    FMainDB: TDatabase;
    { Number of login retries allowed }
    FNumAttemptsAllowed: Integer;
    { Number of failed login attempts so far }
    FNumFailedAttempts: Integer;
    { User's password }
    FPassword: string;
    { Proper first name for user }
    FUserFirstName: string;
    { Proper full name for user }
    FUserFullName: string;
    { Proper last name for user }
    FUserLastName: string;
    { System ID for user }
    FUserID: LongInt;
    { Login username for user }
    FUsername: string;
    FOnLogin: TLoginEvent;          { Event-handler }
    FOnLoggingIn: TLoggingInEvent;  { Event-handler }
    FOnLogout: TNotifyEvent;        { Event-handler }
    FOnLoggingOut: TLoggingOutEvent; { Event-handler }
    FOnBadLogin: TNotifyEvent;      { Event-handler }
    procedure SetMainDB(Value: TDatabase);
  public
    constructor Create(AOwner: TComponent); override;
    procedure Login(UserName, Password: string);
    procedure Logout;
    property ApplicationID: LongInt read FApplicationID
     write FApplicationID;
    property MainDB: TDatabase read FMainDB write SetMainDB;
    property NumAttemptsAllowed: Integer
     read FNumAttemptsAllowed
     write FNumAttemptsAllowed default defNumAttempts;
    property Password: string read FPassword;
    property UserFirstName: string read FUserFirstName;
    property UserFullName: string read FUserFullName;
    property UserLastName: string read FUserLastName;
    property UserID: LongInt read FUserID;
    property Username: string read FUsername;
    property OnLogin: TLoginEvent
      read FOnLogin write FOnLogin;
    property OnLoggingIn: TLoggingInEvent
      read FOnLoggingIn write FOnLoggingIn;
    property OnLogout: TNotifyEvent
      read FOnLogout write FOnLogout;
    property OnLoggingOut: TLoggingOutEvent
      read FOnLoggingOut write FOnLoggingOut;
    property OnBadLogin: TNotifyEvent
      read FOnBadLogin write FOnBadLogin;
  end;
var LoginManager: TLoginManager;
implementation
uses
  Controls, Dialogs, DMLogin;
{ TLoginManager }
constructor TLoginManager.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Establish connection to the data module code }
  LoginDM := TLoginDM.Create(Self);
  FApplicationID := cNoAppID;
  FUserID := cNoUserID;
  FUserFullName := '';
  FUserFirstName := '';
  FUserLastName := '';
  FNumAttemptsAllowed := DefNumAttempts;
end;
procedure TLoginManager.SetMainDB(Value: TDatabase);
var I: Integer;
begin
  if Value <> FMainDB then begin
    FMainDB := Value;
    { Initialize the dataset components in the data module }
    for I := 0 to LoginDM.ComponentCount - 1 do
      if LoginDM.Components[I] is TDBDataSet then
        with TDBDataSet(LoginDM.Components[I]) do begin
          Active := False;
          DatabaseName := FMainDB.DatabaseName;
        end;
  end;
end;
{ Remainder of routines not implemented yet }
initialization
  LoginManager := nil;
  LoginManager := TLoginManager.Create(Application);
end.
```

module (which contains all the data access components supporting TLoginManager) and connects them to the application's TDatabase component via the DatabaseName property. Remember, TLoginManager and the LoginDM data module have no TDatabase of their own, therefore we cannot assign the data module's data components to a database at design time.

## Database Connections

The first thing we need to do is implement enough of Login and Logout to actually connect and disconnect the database. To support this, we'll add four protected methods to TLoginManager as shown in Listing 5. It may seem odd to have two layers of code to connect and two layers to disconnect. However, this technique affords greater flexibility if more than one database is used in the application.

With our two layer approach, ConnectDB is responsible for making the connection for a single TDatabase component (likewise for DisconnectDB). If we have a requirement for multiple databases, we simply add code in Connect and Disconnect for each database. Since we only call Connect and Disconnect throughout TLoginManager to perform these tasks, we have encapsulated the points where we may have to make changes for multiple databases. We'll see exactly how to do this next month.

## The Login Method

Connect and Disconnect are not called by the application, they are internal to TLoginManager. The application only operates through Login and Logout, so let's work on those next. Our first cut at the Login method is shown in Listing 6. All we are concerned with at this point is handling the connection to the database and counting failed login attempts.

The first thing Login does is call Logout to disconnect any existing user from the system. We do this to force any logout-specific actions to occur (like posting to the audit trail that the previous user was logged out). Then we enter a try-except block to trap any exceptions that occur as we attempt to login. This allows us to record and count bad login attempts. We call our Connect method to actually attempt to log into the database with the given username and password. We rely on the RDBMS to validate the login and produce an exception if it fails.

If the login attempt fails, we report the exception back to the user using HandleException. This displays the exception message and removes it from the exception stack. Why do this rather than reraise the exception at the end of our except block? Because additional messages may be forthcoming in our error-handling code and we want to be sure that the original error that caused the login to fail is shown first.

After reporting the failed login, we disconnect from the database. In principle we were never connected to begin with, but if we incorporate the multiple database logic we talked about earlier, we might have connected to one or more databases but failed on another. This way we ensure that we properly disconnect from everything we might be attached to. Also, later as we build up our Login method, we'll see that we might have errors that result in a failed login even after we've successfully connected to the database.

Normally, after a failed login attempt, the application will remain running to allow the user to try again. However, if the number of unsuccessful attempts exceeds the threshold, we terminate the application. The behavior you

➤ *Listing 5*

```
procedure TLoginManager.Connect;
begin
  ConnectDB(FMainDB, FUsername, FPassword);
end;
procedure TLoginManager.ConnectDB(DB: TDatabase; Username, Password: String);
begin
  if DB <> nil then
    with DB do begin
      Connected := False;
      LoginPrompt := False;  { Disable Delphi's login dialog }
      Params.Values['USER NAME'] := Username;
      Params.Values['PASSWORD'] := Password;
      KeepConnection := True;
      Connected := True;
    end;
end;
procedure TLoginManager.Disconnect;
begin
  DisconnectDB(FMainDB);
end;
procedure TLoginManager.DisconnectDB(DB: TDatabase);
begin
  if DB <> nil then
    with DB do begin
      KeepConnection := False;
      Connected := False;
    end;
end;
```

➤ *Listing 6*

```
procedure TLoginManager.Login(UserName, Password: String);
begin
  Logout;     { Logout any existing user }
  FUsername := Username;
  FPassword := Password;
  try
    { Any exception occurring within this block is considered a failed login attempt }
    Connect;
    FNumFailedAttempts := 0;
  except      { Failed login attempt }
    on E: Exception do begin
      Application.HandleException(Self);
      Disconnect;
      Inc(FNumFailedAttempts);
      if FNumFailedAttempts >= FNumAttemptsAllowed then begin
        MessageDlg(IntToStr(FNumFailedAttempts) + ' login attempts have failed.'+
         ' Shutting down the application.', mtError, [mbOk], 0);
        Application.Terminate;
      end;
      FUsername := '';
      FPassword := '';
    end;
  end;
end;
```

incorporate for your system may be more sophisticated. You might not allow the application to be re-entered until a certain amount of time has elapsed.

## The Logout Method

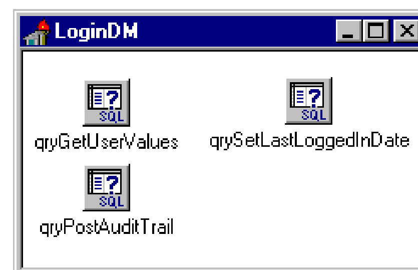
The basic `Logout` method is vastly simpler and is shown in Listing 7. All we have to do is call `Disconnect`. But don't worry, `Logout` will have much more to do later.

## Implementing Data Access

Now we have a rudimentary login manager, but it's no big deal yet. It doesn't do much more than set `TDatabase.Connected`. What's going to make this technique really useful is looking up specific user data and posting events in the database's audit trail. Let's turn our focus now to the data access functions required by `TLoginManager`. The `LoginDM` data module contains all the data components specifically used by `TLoginManager` (see Figure 3) *[If you are working with Delphi 1 you can 'fake' a data module using a regular Delphi form which is never displayed to the user. Editor]*. Remember that these data components are not bound to a database at design time; they get linked to the application's `TDatabase` component via `TLoginManager.SetMainDB`.

Listing 8 shows the complete `LoginDM` data module. If you look at the `Create` method, you'll see that `LoginDM` is tightly bound to `TLoginManager`. `TLoginDM` includes a protected field called `FLogin` of type `TLoginManager` to retain a pointer to `LoginManager`. With this, the data module can directly manipulate the public fields and properties of `TLoginManager`, reducing the amount of data passing required to communicate between these components. Capturing this pointer is accomplished in `TLoginDM.Create` as shown in Listing 8. Since `TLoginManager.Create` passes `Self` as the owner when it in turn calls `TLoginDM.Create` (see Listing 4), then the `AOwner` property in

➤ *Figure 3*




➤ *Listing 7*

```
procedure TLoginManager.Logout;
begin
  if FUserID <> cNoUserID then begin
    Disconnect;
    FUserID := cNoUserID; { Signal that no user is logged in }
  end;
end;
```

➤ *Listing 8*

```
unit DMLogin;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DB, DBTables, Login;
const
  evtLoginSuccessful  = 100;
  evtLoginFail        = 101;
  evtLogout           = 102;
  evtChangePassword   = 103;
type
  TLoginDM = class(TDataModule)
    qryGetUserValues: TQuery;
    qryPostAuditTrail: TQuery;
    qrySetLastLoggedInDate: TQuery;
  protected
    FLogin: TLoginManager;
  public
    constructor Create(AOwner: TComponent); override;
    procedure GetUserValues(var UserID: LongInt;
      var FirstName: String; var LastName: String;
      var DateLastLogin: TDateTime;
      var PasswordExpired: Boolean);
    procedure PostAuditTrail(EventID: Integer;
      EventMsg: String);
    procedure PostUserLoginDate;
  end;
var LoginDM: TLoginDM;
implementation
{$R *.DFM}
constructor TLoginDM.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  if not (AOwner is TLoginManager) then
    raise Exception.Create('TLoginDM cannot be created '+
      'independently of TLoginManager.');
  FLogin := AOwner as TLoginManager;
end;
procedure TLoginDM.GetUserValues(var UserID: LongInt;
  var FirstName: String; var LastName: String; var
  DateLastLogin: TDateTime; var PasswordExpired: Boolean);
{ Returns key information about the current user }
begin
  { Query used:
      SELECT UserID, FirstName, LastName, PasswordLifespan,
             DateLastPasswordChange, DateLastLogin
        FROM Users
        WHERE Username = :Username    }
  with qryGetUserValues do begin
    Close;
    ParamByName('Username').AsString := FLogin.Username;
    Open;
    try
      UserID          := FieldByName('UserID').AsInteger;
      FirstName       := FieldByName('FirstName').AsString;
      LastName        := FieldByName('LastName').AsString;
      DateLastLogin   :=
        FieldByName('DateLastLogin').AsDateTime;
      PasswordExpired := False; { we'll expand this next month }
    finally
      Close;
    end;
  end;
end;
procedure TLoginDM.PostAuditTrail(EventID: Integer;
  EventMsg: String);
{ Writes an entry in the audit trail }
begin
  { Query used:
      INSERT INTO AuditTrail
          (ApplicationID, EventID, Timestamp, UserID,
           Description)
        VALUES
          (:ApplicationID, :EventID, GetDate(),
           :UserID, :Description)
      Note: AuditTrailID field is an auto-increment field }
  with qryPostAuditTrail do begin
    ParamByName('ApplicationID').AsInteger :=
      FLogin.ApplicationID;
    if FLogin.UserID = cNoUserID then
      ParamByName('UserID').Clear
    else
      ParamByName('UserID').AsInteger := FLogin.UserID;
    ParamByName('EventID').AsInteger := EventID;
    ParamByName('Description').AsString := EventMsg;
    ExecSQL;
  end;
end;
procedure TLoginDM.PostUserLoginDate;
{ Writes current date as the user's "date last logged in" }
begin
  { Query used:
      UPDATE Users SET DateLastLogin = GetDate()
        WHERE UserID = :UserID      }
  with qrySetLastLoggedInDate do begin
    ParamByName('UserID').AsInteger := FLogin.UserID;
    ExecSQL;
  end;
end;
end.
```

TLoginDM.Create can be typecast and set in TLoginDM.FLogin to retain a pointer back to LoginManager. Confused? Just keep in mind that FLogin always refers to LoginManager.

One of the jobs of LoginManager is to get some information about the current user from the database. In LoginDM, the query qryGetUserValues looks up the record in the Users table and returns the info we want. The listing shows the GetUserValues procedure which runs this query and passes back the data to LoginManager (since the return fields are protected in TLoginManager, we can't directly write to them from here). We'll deal with expired passwords next month, so for now we'll always set PasswordExpired to False.

The other significant data access required by TLoginManager is to write audit trail records for various login events. Each distinct event that could appear in the audit trail has a unique code and is defined as a constant at the top of the DMLogin unit.

All audit trail events are written to the AuditTrail table through the PostAuditTrail procedure. This procedure encapsulates a query which simply contains a SQL INSERT statement to write the record in the table. Here, we use the GetDate() function provided with Microsoft SQL Server to return the current date and time from the server. If available, it's important to use a server-based function to obtain timestamping information. This ensures a consistent timestamp from a single source no matter how the date and time are set on the individual workstations.

Finally, we'll need to write the current date and time in the Users table to note the most recent login by this user. This is done through the PostUserLoginDate procedure.

## Event Handlers

Event handlers are a basic function of component design. The calling program will register its local event handler procedures with LoginManager (see Listing 1) and LoginManager will simply execute them at the proper points in the process.

## Putting It All Together

Listing 9 shows the completed Login and Logout methods for TLoginManager (for this month), pulling in all the data access and event-handler code we've been discussing.

Figures 4 and 5 show a small demo program that illustrates how all this works.

The Application menu contains items for Login and Logout. Once a user clicks the Application | Login menu item, the application appears as shown in Figure 4. No user is logged in at this point and the status bar at the bottom verifies that. The login dialog overlaying the application is a simple dialog to capture username and password and nothing more: there is no logic behind it.

Once the user clicks OK to actually login, the LoginManager takes over, connects the application to the database, updates the status bar and activates a table to populate the data-aware grid shown in Figure 5. This application is displaying the contents of the AuditTrail table. You can see matched login and logout pairs (events 100 and 102 respectively) for different users. The last line in the audit trail is the login event for the user who made this screen capture.

If the user clicks Application | Logout from the menu, the grid disappears and the status bar returns to the state shown in Figure 4.

The application code to accomplish all this is shown in Listing 10.

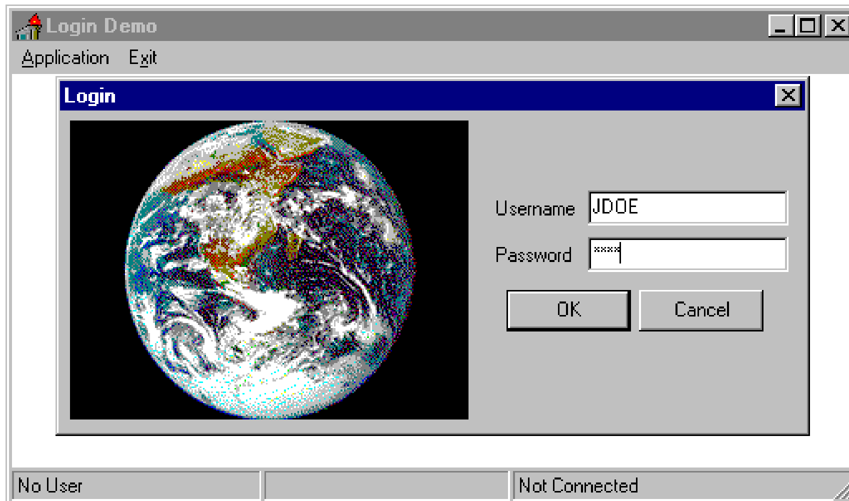
➤ *Listing 9*

```
procedure TLoginManager.Login(UserName, Password: String);
var Cancel: Boolean;
begin
  Logout;
  FUsername := Username;
  FPassword := Password;
  try
    { Any exception occurring within this block is considered a failed login attempt }
    Connect;
    FUserID := cNoUserID;
    LoginDM.GetUserValues(FUserID, FUserFirstName, FUserLastName,
      FDateLastLogin, FPasswordExpired);
    FUserFullName := FUserFirstName + ' ' + FUserLastName;
    if Assigned(FOnLoggingIn) then begin
      FOnLoggingIn(Self, UserName, Password, Cancel);
      if Cancel then begin
        Disconnect;
        Exit;
      end;
    end;
    LoginDM.PostAuditTrail(evtLoginSuccessful, '');
    LoginDM.PostUserLoginDate;
    FNumFailedAttempts := 0;
  except
    on E: Exception do begin
      { Failed login attempt }
      Application.HandleException(Self);
      Disconnect;
      if Assigned(FOnBadLogin) then
        FOnBadLogin(Self);
      Inc(FNumFailedAttempts);
      if FNumFailedAttempts >= NumAttemptsAllowed then begin
        MessageDlg(IntToStr(FNumFailedAttempts) + ' login attempts have failed.'+
          ' Shutting down the application.', mtError, [mbOk], 0);
        Application.Terminate;
      end;
      FUserID := cNoUserID;
      FUsername := '';
      FPassword := '';
      Exit;
    end;
  end;
  if Assigned(FOnLogin) then
    FOnLogin(Self, UserName, Password);
end;
procedure TLoginManager.Logout;
var Cancel: Boolean;
begin
  if FUserID <> cNoUserID then begin
    if Assigned(FOnLoggingOut) then begin
      Cancel := False;
      FOnLoggingOut(Self, Cancel);
      if Cancel then Exit;
    end;
    Disconnect;
    if Assigned(FOnLogout) then FOnLogout(Self);
    LoginDM.PostAuditTrail(evtLogout, '');
    FUserID := cNoUserID;
  end;
end;
```

## Conclusion

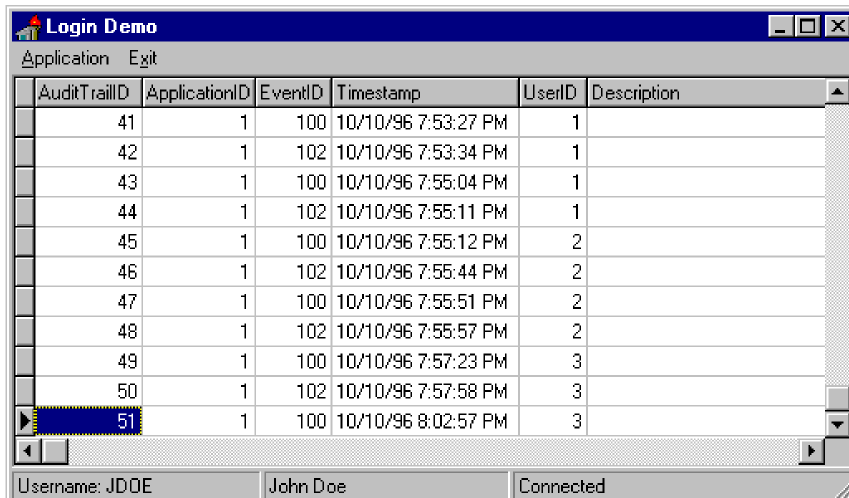
Many database systems may not require this much in the way of managing connections to the database. With more sophisticated or sensitive client/server projects, you may find the need to be more meticulous about user connections. The `TLoginManager` project was intended to show you some of the concerns that may arise in such a project and one technique for handling them.

The main advantage of this object-oriented approach is that it encapsulates all the business rules for user connections in one place. This makes it easier to maintain and extend the rules as the system evolves, particularly if more than one application is connecting to the same database system.




➤ *Above: Figure 4*

➤ *Below: Figure 5*




| AuditTrailID | ApplicationID | EventID | Timestamp | UserID | Description |
|---|---|---|---|---|---|
| 41 | 1 | 100 | 10/10/96 7:53:27 PM | 1 | |
| 42 | 1 | 102 | 10/10/96 7:53:34 PM | 1 | |
| 43 | 1 | 100 | 10/10/96 7:55:04 PM | 1 | |
| 44 | 1 | 102 | 10/10/96 7:55:11 PM | 1 | |
| 45 | 1 | 100 | 10/10/96 7:55:12 PM | 2 | |
| 46 | 1 | 102 | 10/10/96 7:55:44 PM | 2 | |
| 47 | 1 | 100 | 10/10/96 7:55:51 PM | 2 | |
| 48 | 1 | 102 | 10/10/96 7:55:57 PM | 2 | |
| 49 | 1 | 100 | 10/10/96 7:57:23 PM | 3 | |
| 50 | 1 | 102 | 10/10/96 7:57:58 PM | 3 | |
| 51 | 1 | 100 | 10/10/96 8:02:57 PM | 3 | |

## Next Month

A glance at the requirements shown in Figures 1 and 2 show that we're not quite done yet.

Next month, we'll finish up our login manager. We'll discuss more of the practical uses of the login event handlers, post bad login attempts to the audit trail (a neat trick since we can't legally connect to the database), deal with expired passwords, multiple databases and add a few more odds and ends to extend the capabilities of `LoginManager`.


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 10*

```
procedure TfrmMain.LoginManagerLogin(Sender: TObject;
  Username: string; Password: string);
begin
  StatusLine.Panels[0].Text :=
    'Username: ' + LoginManager.Username;
  StatusLine.Panels[1].Text := LoginManager.UserFullName;
  StatusLine.Panels[2].Text := 'Connected';
  tblAuditTrail.Open;
  grdAuditTrail.Visible := True;
  { Enable Application | Logout menu item }
  ApplicationLogoutItem.Enabled := True;
end;
procedure TfrmMain.LoginManagerLogout(Sender: TObject);
begin
  StatusLine.Panels[0].Text := 'No User';
  StatusLine.Panels[1].Text := '';
  StatusLine.Panels[2].Text := 'Not Connected';
  { Disable Application | Logout menu item }
  ApplicationLogoutItem.Enabled := False;
  grdAuditTrail.Visible := False;
end;
procedure TfrmMain.LoginManagerBadLogin(Sender: TObject);
begin
  ShowMessage('That was a bad login. Naughty, naughty.');
end;
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  { Register with LoginManager }
  with LoginManager do begin
    ApplicationID := 1;
    MainDB := dbAppMain;
    OnLogin      := LoginManagerLogin;
    OnLogout     := LoginManagerLogout;
    OnBadLogin   := LoginManagerBadLogin;
  end;
end;
procedure TfrmMain.FormClose(Sender: TObject;
  var Action: TCloseAction);
{ Make sure shutting down the app causes a logout }
begin
  LoginManager.Logout;
end;
procedure TfrmMain.ApplicationLoginItemClick(
  Sender: TObject);
{ Event-handler for Application | Login menu item }
var
  Username, Password: string;
begin
  if LaunchLoginDialog(Username, Password) = mrOK then
    LoginManager.Login(Username, Password);
end;
procedure TfrmMain.ApplicationLogoutItemClick(
  Sender: TObject);
{ Event-handler for Application | Logout menu item }
begin
  LoginManager.Logout;
end;
procedure TfrmMain.ExitMenuClick(Sender: TObject);
begin
  Close;
end;
```