Surviving Client/Server: Managing User Logins, Part 2

by Steve Troxell

Last month, we began development of the `TLoginManager` component which consolidates and centralizes our system login activities across several applications. We built up the component's framework and partially implemented the `Login` and `Logout` functions which simply connected and disconnected the calling application's native `TDatabase` component and posted successful logins and logouts to an audit trail table in our database. This month, we'll continue our work with the `TLoginManager` component by posting unsuccessful logins to the audit trail and detecting and responding to expired passwords.

Our work in this issue follows closely what we started last month and in many cases we'll be revising code we first built in the last issue. To save space, I'll only show partial code listings to point out changes made to previous code.

The specific code changes for this month's discussion are shown in red in the listings. I'll trust you to refer to the previous issue to refresh yourself with what we were doing. The complete code for the `TLoginManager` system (including a demo program) can be found on this month's disk in the SURVIVE directory.

Posting Unsuccessful Logins

Whenever a user logs in or logs out through `TLoginManager`, an audit entry is posted to the `AuditTrail` table. We would also like to post an entry for every unsuccessful login attempt as well. Supposedly, this will reveal attempts to crack an individual's login because several consecutive "bad logins" will appear for the same username. The problem we face with this is that if we don't have a valid login, how can we write something to the database?

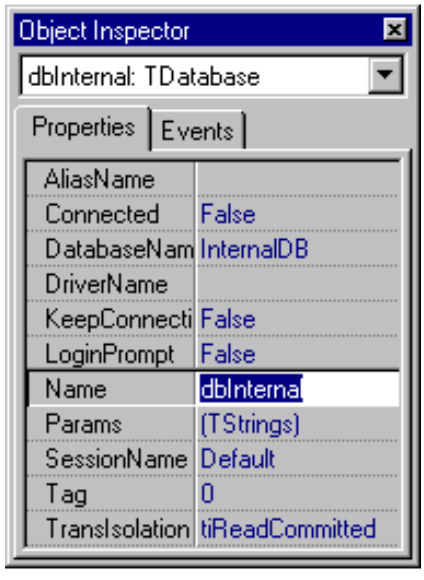
The solution we'll devise provides a fixed user account with which our applications could connect to the database for their own purposes independently of the user supplied login information. This account would have a username that would not conflict with a valid user of the system; for example something gibberish or something blatantly proprietary like "SYSTEM". Also, the password would not be known by the customers, only the developers.

Having a separate system account opens the possibility for additional layers of data security. We'll be using the system account to send queries posting audit trail entries. Because of this, we could revoke write permissions to the `AuditTrail` table from all normal users and only grant them to the system account. This way users cannot log in through ISQL or ODBC and modify the audit trail, but your applications can post records there unimpeded. Note that this can also be achieved by using a stored procedure to write to the audit trail since they can access database tables independently of the permissions of the user that executed them (see Issue 6, February 1996).

To implement the system account, we'll add a `TDatabase` component in the `LoginDM` data module with the properties shown in Figure 1. Next we'll attach the `qryPostAuditTrail` query component to this database by setting its `DatabaseName` property to `InternalDB`. We'll want to assign the login account at runtime in the `TLoginManager.Create` method as shown in Listing 1.

In this case `SystemAccountUserName` and `SystemAccountPassword` are simply constants within the data module for the sake of illustration. In a production system, you would want to externalize these values; possibly by supplying them through a DLL. In this way you can take more elaborate steps to encrypt the account information and also leave open the possibility of changing the system account

➤ *Figure 1*



➤ *Listing 1*

```
const
  SystemAccountUserName = 'SYSTEM';
  SystemAccountPassword = 'onomatopoeia';
  . . .
constructor TLoginDM.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  . . .
  with dbInternal do begin
    Params.Values['USER NAME'] := SystemAccountUserName;
    Params.Values['PASSWORD'] := SystemAccountPassword;
  end;
end;
```

without having to re-compile your applications.

Finally, we need to associate the TDatabase component with the actual database. This obviously would be the same database that the user is attempting to log into, so we'll just intervene when the calling application registers its TDatabase component by setting the MainDB property (see Listing 2). We'll also have to adjust our logic that loops through all the data module components and links them to the application's TDatabase. We specifically linked our qryPostAuditTrail query to the internal database component at design time and we don't want it to be overridden at runtime.

Now, whenever we post an entry in the audit trail through the qryPostAuditTrail query, we will open a new database connection with the system account, write the record, then close the connection. The final touch is to add the call to post the audit entry when an unsuccessful login is made as shown in Listing 3. Notice that we trap and discard any exceptions that might be raised during the posting of the bad login entry to the audit trail. This is done to cover the possibility that the unsuccessful login might be the result of the database server itself being down or unavailable. In that case, the posting of the audit trail entry would be impossible and would itself produce an exception.

Changing User Passwords

One of the features of the login system we have developed is that we can require any individual user's password to expire after a given number of days have elapsed. The "lifespan" of the password can vary from user to user if we so desire. To accommodate this, for each user we store the interval of their password lifespan in days and the date of their last password change. With this, our login manager can detect when passwords have expired and require that the user change their password before completing the login. In addition, we would like to allow the user to voluntarily change their password at any time.

Some backend products have this password functionality built in, some don't. Even for those that do, your users are likely to see the backend vendor's error messages and dialogs. By implementing this functionality ourselves, we provide the features even if the backend vendor does not, and we present a consistent, seamless interface to the users.

However, keep in mind that we are relying on the RDBMS to store and validate usernames and passwords when we connect to the database, so we must interface to the RDBMS to actually perform the password change. Exactly how this is done will vary greatly from one backend to another. For that reason, we will implement the mechanics of changing the user's password within a DLL which we could customize for any particular backend we wanted.

Microsoft SQL Server allows passwords to be changed via the sp_password system procedure. In addition, we must change the value of DateLastPasswordChange for the

➤ *Listing 2*

```
procedure TLoginManager.SetMainDB(Value: TDatabase);
var I: Integer;
begin
  if Value <> FMainDB then begin
    FMainDB := Value;
    LoginDM.dbInternal.AliasName := FMainDB.AliasName;
    { 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
          if DatabaseName = '' then begin
            Active := False;
            DatabaseName := FMainDB.DatabaseName;
          end;
        end;
  end;
end;
```

➤ *Listing 3*

```
procedure TLoginManager.Login(UserName, Password: String);
var Cancel: Boolean;
begin
  Logout;
  FUsername := Username;
  FPassword := Uppercase(Password);
  try
    Connect;
    {. . . skip a bunch of junk here }
  except
    on E: Exception do begin     { Failed login attempt }
      Application.HandleException(Self);
      Disconnect;
      if Assigned(FOnBadLogin) then
        FOnBadLogin(Self);
      { Post bad login event }
      try
        LoginDM.PostAuditTrail(evtLoginFail, 'Username: ' + FUsername);
      except
      end;
      {. . . skip some more junk here }
    end;
  end;
end;
```

➤ *Listing 4*

```
create procedure ChangePassword(
  @Username    varchar(30),
  @OldPassword varchar(30),
  @NewPassword varchar(30))
as
begin
  declare @Result integer
  execute @Result = sp_password @OldPassword, @NewPassword
  if @Result <> 0
  begin
    raiserror 50001 "Could not change user password"
    return
  end
  update Users
    set DateLastPasswordChange = getdate()
    where Username = @Username
end
```

➤ *Listing 5*

```
{ DLL Project File: }
library Password;
uses Pass1;
exports ChangePassword;
end.

{ DLL Unit File: }
unit Pass1;
interface
uses
  SysUtils, DB, DBTables;
function ChangePassword(iAliasName    : PChar;
                        iUserName     : PChar;
                        iOldPassword  : PChar;
                        iNewPassword  : PChar;
                        var oErrMsg   : PChar): Word; export;
implementation
uses
  PassInt;
type
  ESamePassword = class(Exception);
  EErrorChangingPassword = class(Exception);
function ChangePassword;
var
  TempDatabase: TDatabase;
  TempQuery: TQuery;
begin
  Result := cpSuccess;
  StrPCopy(oErrMsg, '');
  try
    { Validate the new password }
    if StrIComp(iOldPassword, iNewPassword) = 0 then
      raise ESamePassword.Create('');
    { Create a TDatabase structure to connect with }
    TempDatabase := TDatabase.Create(nil);
    try
      with TempDatabase do begin
        AliasName := StrPas(iAliasName);
        DatabaseName := 'PasswordChangeDB';
        Params.Values['USER NAME'] := StrPas(iUserName);
        Params.Values['PASSWORD'] := StrPas(iOldPassword);
        LoginPrompt := False;
        Connected := True;
      end;
      { Create a query to run the system procedure with }
      TempQuery := TQuery.Create(nil);
      with TempQuery do begin
        try
          DatabaseName := TempDatabase.DatabaseName;
          { Change the password in the RDBMS and set date of change }
          SQL.Add(Format('execute ChangePassword %s, %s, %s', [StrPas(iUserName),
            StrPas(iOldPassword), StrPas(iNewPassword)]));
          try
            ExecSQL;
          except
            raise EErrorChangingPassword.Create('');
          end;
        finally
          Free;
        end;
      end;
    finally
      TempDatabase.Free;
    end;
  except
    on ESamePassword do begin
      Result := cpSamePassword;
      StrPCopy(oErrMsg, 'New password cannot be the same as the old password');
    end;
    on EErrorChangingPassword do begin
      Result := cpErrorChangingPassword;
      StrPCopy(oErrMsg, 'Error changing password');
    end;
    on E: Exception do begin
      Result := cpUnknown;
      StrPCopy(oErrMsg, E.Message);
    end;
  end;
end;
end.
```

➤ *Listing 6*

```
unit PassInt;
interface
const
  cpSuccess = 0;
  cpErrorChangingPassword = 1;
  cpSamePassword = 2;
  cpUnknown = 999;
function ChangePassword(AliasName : PChar; UserName : PChar; OldPassword : PChar;
  NewPassword : PChar; var ErrMsg : PChar): Word; external 'PASSWORD.DLL';
implementation
end.
```

current user in our Users table. We'll encapsulate all this within a stored procedure as shown in Listing 4. Listing 5 shows the source for the DLL (PASSWORD.DLL) which we'll use to call our stored procedure. Listing 6 shows the interface unit we'll use to communicate with this DLL from the main application.

In Listing 5, you'll see we're passing in the alias name for the database we're connected to and creating an independent connection from within the DLL. We can't simply pass our existing `TDatabase` component into the DLL because referencing an application's VCL objects from within a DLL can create nasty problems. The user's old password is supplied as well: partly because some backend products require this to perform a password change and partly to allow the DLL to centralized the business rules for changing passwords. For example, users should not be allowed to change their password to the same value as their existing password.

The next step is to integrate the DLL with `TLoginManager`. Listing 7 shows a new public method called `ChangePassword`. This method can be called from the application at any time to allow for voluntary password changes. Notice that we call `TLoginManager.Connect` after changing the password. This terminates our current database connection and re-establishes a new one under the new password. This is necessary since Delphi may later attempt to open additional database connections for multiple concurrent datasets, which would fail because the password stored within the `TDatabase.Params` property would still be the old password. The `Connect` method ensures the correct password is available.

Because the `TDatabase` component is momentarily disconnected, any datasets that are active at the time will be closed. You'll have to guard against this when calling `ChangePassword`.

Now we must provide a dialog with which the user may enter a new password. Figure 2 and Listing 8 show the dialog unit which is

compiled into our main application. For the sake of brevity, I have omitted the `interface` section for this unit. It simply declares a form class called `TdlgPassword` with no special methods or properties. The standalone procedures `LaunchChangePasswordDialog` and `LaunchPasswordExpiredDialog` are exported by the unit. These are the routines which our application calls to show this dialog; in the case of an expired password, we only change the dialog caption. Note that the dialog itself is responsible for calling `TLoginManager.ChangePassword` to effect the change, and that the dialog does not close unless the password change completes successfully or the user explicitly cancels it.

Now the main application has only to call `LaunchChangePasswordDialog` to provide a voluntary change password feature.

Expired Passwords

Dealing with expired passwords is a bit more involved. First we must detect that a user's password has expired. This is done in the `LoginDM` data module's `GetUserValues` method. Last month, we simply hard-coded the `PasswordExpired` parameter to `False`. The correct logic is shown in Listing 9 where we examine the `DateLastPasswordChange` and `PasswordLifespan` fields from the `Users` table. We've allowed for a `null` in `DateLastPasswordChange` to force a new password (for example, when the user account is first set up by the administrator, the user would be given a temporary password which must be changed immediately upon first login). Also, a `null` value in the `PasswordLifespan` field indicates the password never expires for this user.

When `PasswordExpired` is `True`, we must signal the calling application to display its "change password" dialog, so we'll add an event handler (see Listing 10). A partial listing of `TLoginManager.Login` is shown in Listing 11. As you can see, immediately after calling `GetUserValues`, we check to see if the password has expired. If so, we fire the event handler and let the application allow the user to enter a new password

➤ *Figure 2*


```
procedure TLoginManager.ChangePassword(OldPassword, NewPassword: string);
var StatusText: PChar;
begin
  if Uppercase(OldPassword) <> FPassword then
    raise Exception.Create(
      'Unable to change password – current password incorrect.');
  StatusText := StrAlloc(255);
  try
    if PassInt.ChangePassword(PChar(FMainDB.AliasName), PChar(FUsername),
      PChar(OldPassword), PChar(NewPassword), StatusText) <> 0 then
      raise Exception.Create(StrPas(StatusText));
    { Set the new password }
    FPassword := Uppercase(NewPassword);
    { Reconnect database(s) with new password }
    Connect;
    { Post a "change password" in the audit trail }
    LoginDM.PostAuditTrail(evtChangePassword, '');
  finally
    StrDispose(StatusText);
  end;
end;
```

➤ *Listing 7*

```
unit fmPasswd;
. . . interface omitted . . .
implementation
{$R *.DFM}
uses
  Login;

function LaunchChangePasswordDialog;
begin
  dlgPassword.Caption := 'Change Password';
  Result := dlgPassword.ShowModal;
end;

function LaunchPasswordExpiredDialog;
begin
  dlgPassword.Caption := 'Password Expired';
  Result := dlgPassword.ShowModal;
end;

procedure TdlgPassword.btnOKClick(Sender: TObject);
begin
  if edtNewPassword.Text <> edtConfirmPassword.Text then
    raise Exception.Create('The confirmation password is not correct.');
  LoginManager.ChangePassword(edtOldPassword.Text, edtNewPassword.Text);
  ModalResult := mrOK;
end;

procedure TdlgPassword.FormShow(Sender: TObject);
begin
  edtOldPassword.Text := '';
  edtNewPassword.Text := '';
  edtConfirmPassword.Text := '';
  edtOldPassword.SetFocus;
end;

end.
```

➤ *Listing 8*

```
{ if DateLastPasswordChange is null, then password change forced }
if FieldByName('DateLastPasswordChange').IsNull then
  PasswordExpired := True
else
  { if PasswordLifespan is null, then password never expires }
  if FieldByName('PasswordLifespan').IsNull then
    PasswordExpired := False
  else
    PasswordExpired := Date - FieldByName('DateLastPasswordChange').AsDateTime >=
      FieldByName('PasswordLifespan').AsInteger;
```

➤ *Listing 9*

```
{ New Event Handler: }
TPasswordExpiredEvent =
  procedure (Sender: TObject; var Cancel: Boolean) of object;

{ Declared in TLoginManager: }
type
  TLoginManager = class(Tcomponent)
  protected
    . . .
    FOnPasswordExpired: TPasswordExpiredEvent;
  public
    . . .
    property OnPasswordExpired: TPasswordExpiredEvent
      read FOnPasswordExpired write FOnPasswordExpired;
  end;
```

➤ *Listing 10*

```
procedure TLoginManager.Login(UserName, Password: String);
var
  Cancel: Boolean;
begin
  . . .
  FUsername := Username;
  FPassword := Uppercase(Password);
  Connect;
  LoginDM.GetUserValues(FUserID, FUserFirstName, FUserLastName,
                        FDateLastLogin, FPasswordExpired);
  FUserFullName := FUserFirstName + ' ' + FUserLastName;
  { Determine if user's password has expired... }
  if FPasswordExpired then begin
    Cancel := True;
    if Assigned(FOnPasswordExpired) then
      FOnPasswordExpired(Self, OldPassword, NewPassword, Cancel);
    if Cancel then
      raise Exception.Create(
        'Unable to login—user''s password has expired');
  end;
  . . .
end;
```

➤ *Listing 11*

```
procedure TfrmMain.LoginManagerPasswordExpired(Sender: TObject;
  var Cancel: Boolean);
begin
  Cancel := LaunchPasswordExpiredDialog <> mrOK;
end;
```

➤ *Listing 12*

```
with qryGetUserValues do begin
  Close;
  ParamByName('Username').AsString := FLogin.Username;
  Open;
  try
    if Eof then
      raise Exception.Create('Invalid login.');
    . . .
```

➤ *Listing 13*

(Listing 12). If they don't, then we terminate the login process. Remember that we have an exception handler within the `Login` method which will disconnect the database if an exception is raised.

Errata

There was a small oversight in last month's code. In the data module's `GetUserValues` method, if the username used to log in is not found in the `Users` table, it should be treated as an unsuccessful login. To correct this, we simply check for `EOF` after calling `qryGetUserValues.Open` and raise an exception as shown in Listing 13.

Next Month

`TLoginManager` is really taking shape now! Next month, we'll finish our work by allowing runtime overrides for the database and server defined in the alias. This is very useful in environments where multiple databases or servers may be employed for development, testing, customer demos, etc. Also, we'll see what's involved in handling multiple databases from a single application.

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