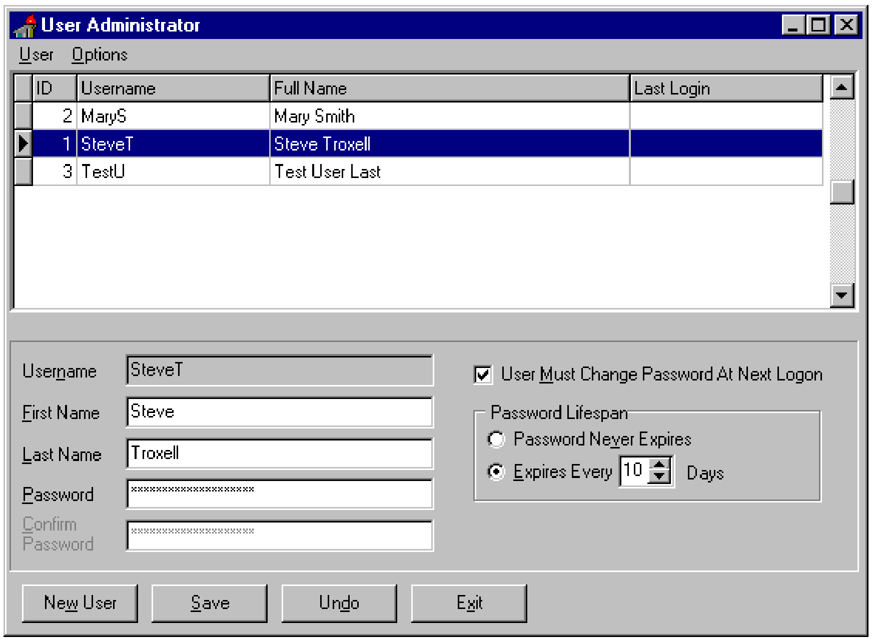
Surviving Client/Server: Customized User Administration

by Steve Troxell

For the past few issues we've been developing a centralized login manager to consistently handle many aspects of a client application's connection to a database. This allowed us to easily add features such as posting login/logout audit trails, having passwords expire and allowing users to change passwords from the client app. Part of the appeal of this technique is that we can provide certain user and password management features in our applications even if those features aren't built into the RDBMS we've chosen. To achieve this, we must store some extra information about the users in our own database table. Of course, when we add users through our RDBM system's user utility, it's not going to update our supplementary data about the users because it has no knowledge of this table. This month, we're going to develop our own user administration utility which adds users both in the RDBMS and in our supplementary data table at the same time.


User Administrator

User Options

ID	Username	Full Name	Last Login
2	MaryS	Mary Smith	
1	SteveT	Steve Troxell	
3	TestU	Test User Last	

Username	SteveT
First Name	Steve
Last Name	Troxell
Password	********************
Confirm Password	********************

[✓] User Must Change Password At Next Logon

Password Lifespan
- ○ Password Never Expires
- [✓ ● Expires Every 10 Days]

```
CREATE TABLE Users(
  UserID             integer,    /* 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 & time of last password change */
  PasswordLifespan smallint)  /* Number of days between forced password changes */
```

➤ *Listing 1*

The User Administrator App

The User Administrator program is shown in the screenshot. This is a minimal version, designed to let us view all the users in the system, add new users and allow us to change any of the user information (including their password). As a reminder from past months' work, the supplementary user table we've used for `TLoginManager` is shown in Listing 1. Our application merely changes this table, but must also add user names and passwords, and change the password for a given user in the RDBMS itself. The complete User Administrator program can be found on this month's disk in the SURVIVE directory (of course, you'll have to adjust its database connection and code for your particular RDBMS).

Adding Users

Integrating with the RDBMS is problematic: your RDBMS is going to have to provide some external means of adding users and changing their password. Many times this is provided via the system's API, usually involving a sequence of calls to a DLL. The details of how your particular RDBMS provides this functionality will vary greatly from vendor to vendor. You'll have to check the documentation. Microsoft SQL Server is the basis for our work this month and SQL Server, in addition to the API, provides SQL commands for adding users and changing passwords.

Listing 2 shows a stored procedure we'll use to add a user. In SQL Server, users must be added at two levels: a user login is added to the SQL Server service and the user must be added specifically to the database itself. In this way, some users could be allowed into certain databases on the server but not others. We use the `sp_addlogin` system stored procedure to add the user and their password to SQL Server and the `sp_adduser` system stored procedure to allow that user to access our database. Then it is a simple matter to add a new row to our `Users` table.

We'll encapsulate the call to this stored procedure into a routine we can call from our Delphi app as shown in Listing 3. This routine is kept isolated from the rest of the app (including dynamically building its own `TQuery` component) so that any changes needed to support different RDBM systems are centralized. If we needed to call a DLL to add users to our database, only this routine would need to be modified, the calling program wouldn't care exactly how users are added to the system.

With this foundation in place, writing the actual application is fairly straightforward. The only special part is the password. Since we don't store the user's password we never actually display it in the Password edit control. We only display an arbitrary number of spaces (shown as the special password mask character). It's typical practice for a user's password to be kept secure even from the user administrator. If the user forgets their password, the administrator simply assigns them a new one and usually requires them to change it immediately upon their next login.

When we add a new user, we clear the data entry fields at the bottom of the screen and set the checkbox and radio controls to the right according to how our system defaults these values. It would be handy to store the default settings in a table as well, as shown in Listing 4 (this begs for a Configuration dialog in the application to set these default values).

➤ Listing 2

```
create procedure AddUser(
  @Username   varchar(30),
  @Password   varchar(30),
  @FirstName  varchar(20),
  @LastName   varchar(20),
  @PasswordLifespan smallint,
  @PasswordMustChange char(1))    /* Y/N */
as
  declare @Result integer
  declare @DateLastPasswordChange datetime
begin
  /* add a server login */
  execute @Result = sp_addlogin @Username, @Password
  if @Result = 0
  begin
    /* add the user to our database */
    execute @Result = sp_adduser @Username
    if @Result <> 0
    begin
      execute sp_droplogin @Username
      return
    end
    if @PasswordLifespan <= 0
      select @PasswordLifespan = null
    if Upper(@PasswordMustChange) = 'Y'
      select @DateLastPasswordChange = null
    else
      select @DateLastPasswordChange = GetDate()
    /* build a record for the supplementary user table */
    insert Users(Username, FirstName, LastName,
                 PasswordLifespan, DateLastPasswordChange)
      values (@Username, @FirstName, @LastName,
              @PasswordLifespan, @DateLastPasswordChange)
    /* if an error, then remove the user from the system */
    if @@error <> 0
    begin
      execute sp_dropuser @Username
      execute sp_droplogin @Username
    end
  end
end
```

➤ Listing 3

```
function DelimitedStr(S: string): string;
begin
  Result := '''' + S + '''';
end;
function BoolToChar(B: Boolean): Char;
begin
  if B then Result := 'Y' else Result := 'N';
end;
procedure AddDBUser(DB: TDatabase; Username, Password, FirstName,
  LastName: string; MustChangePassword: Boolean; PasswordLifespan: Integer);
begin
  with TQuery.Create(nil) do
  try
    DatabaseName := DB.DatabaseName;
    SQL.Add('execute AddUser ');
    SQL.Add(DelimitedStr(Username) + ',');
    SQL.Add(DelimitedStr(Password) + ',');
    SQL.Add(DelimitedStr(FirstName) + ',');
    SQL.Add(DelimitedStr(LastName) +',');
    SQL.Add(IntToStr(PasswordLifespan) + ',');
    SQL.Add(BoolToChar(MustChangePassword));
    ExecSQL;
  finally
    Free;
  end;
end;
```

➤ Listing 4

```
create table AdminConfig(
  PasswordLifespan       smallint null,
  MustChangeNewPassword  char(1) not null default 'N'
    check (MustChangeNewPassword in ('Y', 'N')))
```

User Security

Generally, a RDBM system won't allow just any user to add more users to the system. Nor are they likely to allow just any user to change another user's password. Typically, a program is required to be logged in through the "database administrator" account to permit these actions (we'll call this the DBA account). Therefore, the User Administrator program should connect to the database using the DBA account.

You could hardcode the DBA account's username into the application (through the TDatabase.Params property for example) and require users to enter only a password to gain access to the program. The password would have to match the DBA account's password for the program to successfully connect to the database.

An alternative would be to prepopulate the Users table with a record corresponding to the DBA account's username. This would allow you to use TLoginManager to let users login to User Administrator. It would be simple to add a property to TLoginManager which compares the login user name with the DBA account to let you know if it was the DBA account who was logging in. By having a Users record for the DBA account, you could even use User Administrator to change the password of the DBA account if you so desire.

Editing A User

Allowing user information to be edited is slightly easier. We only have to integrate with the RDBMS if the password changes, the rest is simply a matter of changing our `Users` table. Since we don't actually display the user's current password, it's easy to detect when it has changed: whenever there's non-blank data in the `Password` edit box.

Listing 5 shows the stored procedure we'll use to edit a given user's record. If we detect a new password, we use the built-in `sp_password` system stored procedure to actually change the password. SQL Server allows the system administrator account to change the password of any user and does not require the old password to do so. Therefore, we pass `null` for the old password. Note we must be careful to identify the user for which we're changing the password. Otherwise, we might inadvertently change the password of the DBA account (or whatever account was used to login the User Administrator application).

Just as we encapsulated the `AddUser` stored procedure in an `AddDBUser` Delphi procedure, so we'll encapsulate the `EditUser` stored procedure. The `EditDBUser` Delphi procedure is much the same as `AddDBUser`, so I won't repeat it here. The important difference is that it makes sure to trim spaces from the password before sending it off to the stored procedure.

For this example, I've put the `AddDBUser` and `EditDBUser` Delphi procedures into a unit called `UserAPI`. This unit is now responsible for whatever server-specific code is necessary to implement our interface to the RDBMS. If we need to make DLL calls rather than SQL statements, we need only concern ourselves with `UserAPI`. The rest of the application is not concerned with exactly how the RDBMS integration takes place.

Enhancements

With the User Administrator utility and the `TLoginManager` class, we can add a whole slew of new user management functionality to our applications, whether our RDBMS supports it or not. Some features that we could implement are:

➢ ***Account Lockouts.*** A flag set on a user which prevents any system logins until the flag is cleared. This is a high security measure for when an employee goes on vacation, leave of absence, or just extended out of town business and ensures that their account cannot be used until reactivated. This could be made as elaborate as you wish, to include scheduling user access to specific times of day (for example, the account can only be used between 9am and 5pm Monday through Friday).

➢ ***Password Histories.*** The system could remember a fixed number of passwords for a given user and not allow them to be reused when the user changes their password. Like the password lifespan, the number of passwords to remember (and whether this feature is even in use) could be different for each user.

➢ ***Disallow Password Changes.*** The user would not be allowed to change their password. This feature would aid in the use of shared login accounts such as guest accounts (most likely you would have the password never expire in this case as well).

Conclusion

The point of this exercise is that the login management system we've developed over the last several months permits you to customize the user management features of your software products without being bound by the functionality provided by the RDBMS you've chosen. This self-contained system also greatly facilitates the development and deployment of turnkey systems.

Of course, an extension of "rolling your own" user administration system is providing a means of controlling the access to various modules and screens in the application. But we'll come back to this in a future issue: we've beaten this topic into the ground enough.

Next month we'll see how to write our own custom extensions to the SQL language.

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

➤ *Listing 5*

```
create procedure EditUser(
  @Username   varchar(30),
  @Password   varchar(30),
  @FirstName  varchar(20),
  @LastName   varchar(20),
  @PasswordLifespan smallint,
  @PasswordMustChange char(1))   /* Y/N */
as
  declare @Result integer
  declare @DateLastPasswordChange datetime
begin
  if @Password <> ''
  begin
    execute @Result = sp_password null, @Password, @Username
    if @Result <> 0
    begin
      raiserror 50001 "Could not change user password"
      return
    end
  end
  if @PasswordLifespan <= 0
    select @PasswordLifespan = null
  if Upper(@PasswordMustChange) = 'Y'
    select @DateLastPasswordChange = null
  else
    select @DateLastPasswordChange = GetDate()
  /* change the Users record */
  update Users set
    FirstName = @FirstName,
    LastName = @LastName,
    PasswordLifespan = @PasswordLifespan,
    DateLastPasswordChange = @DateLastPasswordChange
    where Username = @Username
end
```