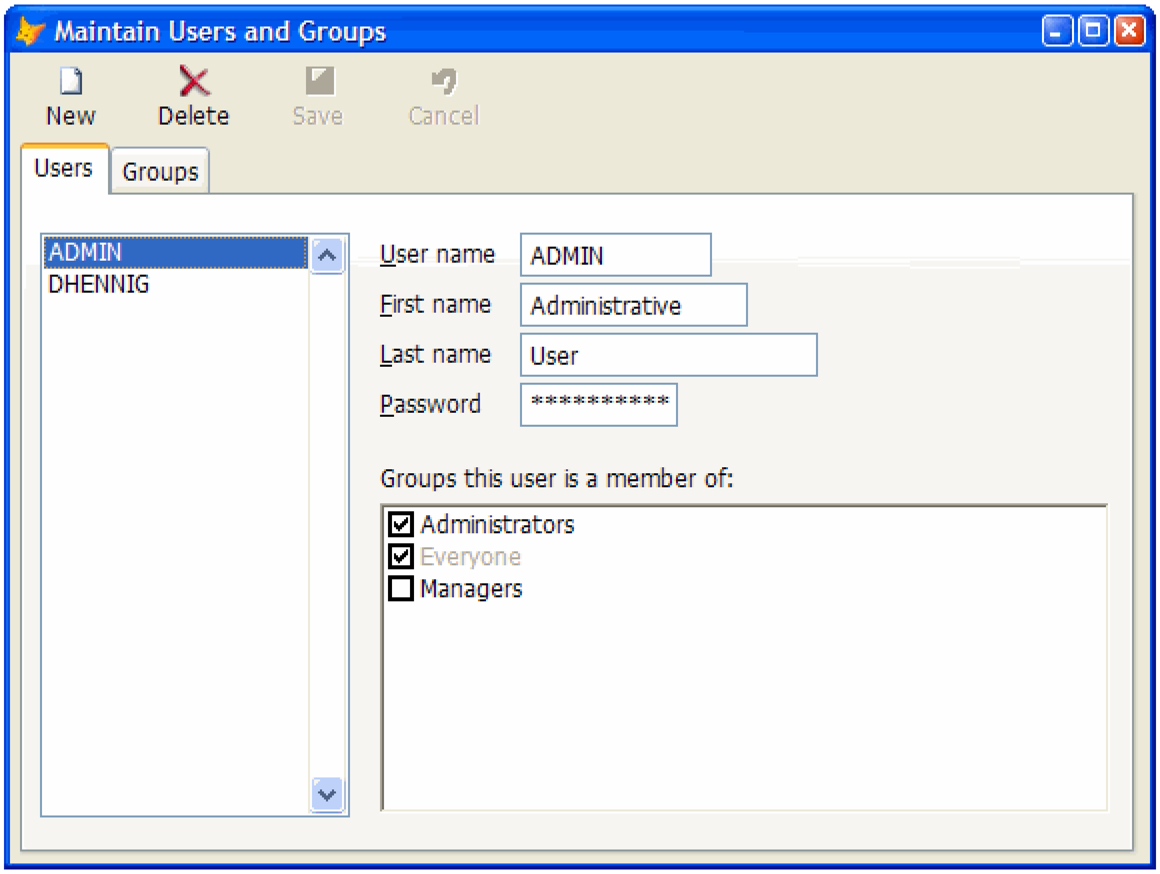
Role-Based Security, Part III

Doug Hennig

This month, Doug Hennig winds up his series on role-based security by examining a form class responsible for maintaining users and roles.

In my previous two articles, I discussed issues related to role-based security and provided several classes that together, make up a framework for application security. In this, the final article in the series, I'll show you a form class the administrators of your applications will use to manage which users can access the application and what roles they belong to.

Before we look at this form class, you may want to run the sample application provided with this month's subscriber downloads. Run MAIN.PRG and, as prompted, login as the ADMIN user. Choose the Maintain Users and Groups function from the File menu. The dialog shown in **Figure 1** and **Figure 2** show this form in action.



User name: ADMIN
First name: Administrative
Last name: User
Password: **********

Groups this user is a member of:

- [x] Administrators
- [x] Everyone
- [ ] Managers

Figure 1. The Users page of SFUsersForm allows an administrator to define what users can access the application and what roles they belong to.

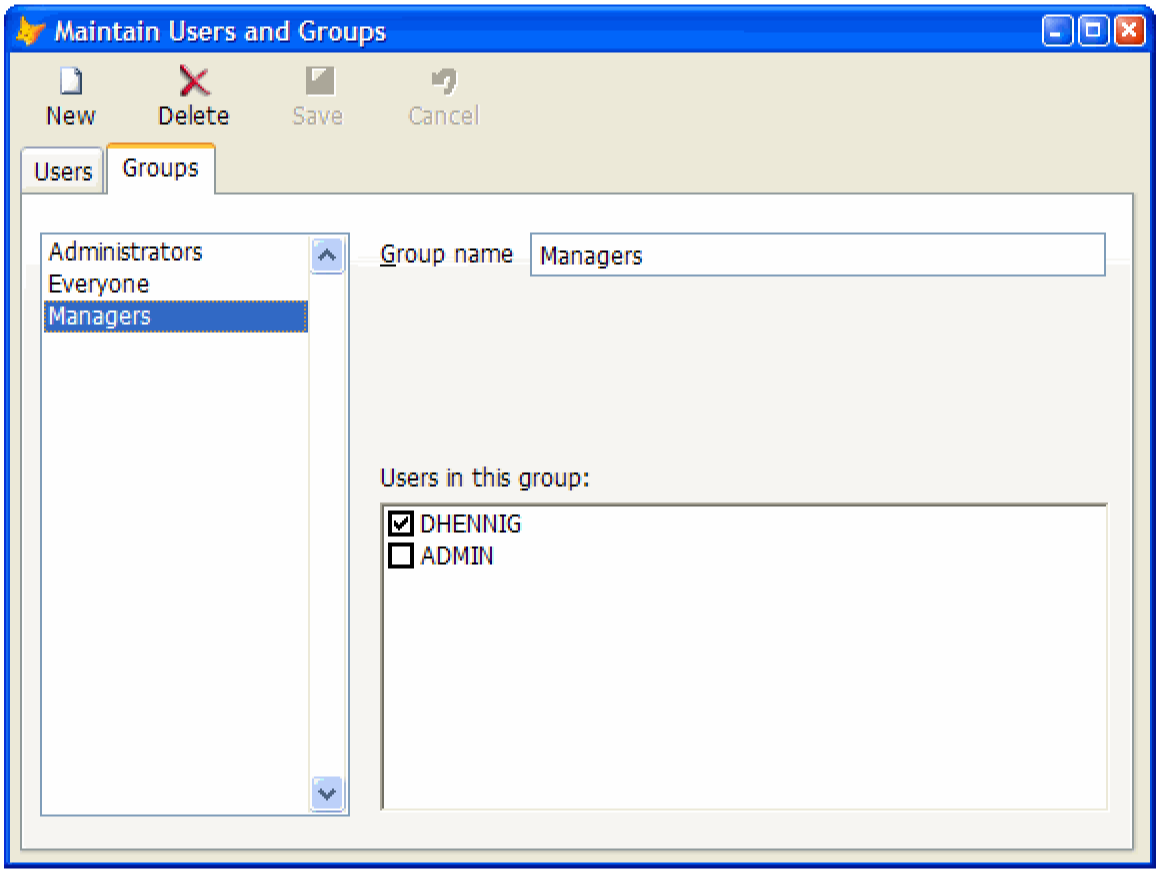


Figure 2. Define user groups and their members on the Groups page of SFUsersForm.

The form is based on the SFUsersForm class in SFSecurityUI.VCX. It consists of a "toolbar" (it isn't a real toolbar; it's just a row of buttons on the form) and a pageframe. The pageframe has two pages: one for users and one for roles. Note that roles are referred to as "groups" in the user interface elements of the form; our users found "groups" easier to understand than "roles", but you can easily change the wording to something else if you wish.

Both pages are similar: they contain a listbox of the defined items, controls to edit attributes of the selected item, and a TreeView. In the case of the Users page, the TreeView shows the defined roles, with checkmarks in front of the roles the selected user belongs. For the Groups page, the TreeView shows the defined users, with checkmarks in front of the members of the selected role. Thus, you can specify role membership from the Users page (select a user and check or uncheck the roles the user belongs to) or from the Groups page (select a role and check or uncheck the members of the role).

The textboxes on the Users page are bound to properties of the oUser member of the form. For example, the User Name textbox is bound to Thisform.oUser.UserName. oUser contains a user object for the selected user; we'll see how oUser references the correct object later.

The Refresh methods of the toolbar buttons and listboxes disable the controls under certain conditions. For example, the New button is only enabled if the form's lCanAdd property is .T. and the listboxes are only enabled if lCanNavigate is .T. We'll see how these properties are controlled later.

Note that although security is actually stored in tables (USERS.DBF, ROLES.DBF, and USERROLES.DBF in the case of user, roles, and the members of each role), SFUsersForm doesn't access these tables directly. Instead, it uses methods of the security manager class I discussed in the previous articles in this series to add, edit, and delete users and roles.

Selecting a user or role

Listboxes are provided on the Users and Groups page as a navigation mechanism. Each listbox uses an internal array property, aItems, as its RecordSource. This array is filled in the Requery method by calling the GetArray method of the oUsers or oRoles member of the security object. Requery has some logic to ensure that if an item was previously selected in the list, it will be reselected when the listbox is requeried. If no item was selected, Requery sets ListIndex to 1. Requery then calls the SelectUser or SelectRole method to set up the form to work with the chosen user or role. Here's the code for lstUsers.Requery; lstRoles.Requery is nearly identical.

```
lparameters tlNoAction
local liUser, ;
  laItems[1], ;
  lnItems, ;
  lnIndex
with This
  if not tlNoAction
    liUser  = iif(vartype(Thisform.oUser) = 'O', ;
      Thisform.oUser.ID, 0)
    lnItems = oSecurity.oUsers.GetArray(@laItems)
    asort(laItems)
    dimension .aItems[lnItems, 2]
    acopy(laItems, .aItems)
    lnIndex = ascan(laItems, liUser, -1, -1, 2, 8)
    if lnIndex = 0 and between(.ListIndex, 1, lnItems)
      lnIndex = ascan(laItems, ;
        .aItems[.ListIndex, 2], -1, -1, 2, 8)
    endif lnIndex = 0 ...
    dodefault()
    do case
      case .ListCount = 0
      case lnIndex > 0 and lnIndex = .ListIndex
        Thisform.SelectUser(.aItems[.ListIndex, 2])
      case liUser = 0 or lnIndex = 0
        .ListIndex = 1
        Thisform.SelectUser(.aItems[.ListIndex, 2])
      otherwise
        .ListIndex = lnIndex
        Thisform.SelectUser(liUser)
    endcase
  endif not tlNoAction
endwith
```

As you may expect, SelectUser and SelectRole are very similar, so we'll just look at SelectUser. SelectUser passes the ID of the chosen user to the GetUserByID method of the oUsers member of the security object and places the returned user object into the oUser property of the form. It then calls the GetRolesForUser method of the security object to return a collection of roles the specified user is a member of. SelectUser adds a node to the TreeView for each defined role and uses the collection of roles the user is in to determine whether to turn on the Checked property for each node. Also, since we don't want the user to uncheck the Everyone role, we'll make it look disabled by setting its ForeColor property appropriately. Unfortunately, there isn't a way to disable individual nodes of a TreeView, so the best we can do is make it look disabled and then handle what happens if the user disregards that and unchecks the node anyway (we''l see that when we look at the Save method later). Finally, since the textboxes are bound to properties of the form's oUser member, SelectUser refreshes the form.

```
lparameters tiUserID
local loRoles, ;
  loRole, ;
  loNode

* Get an object for the specified user.

This.oUser = oSecurity.oUsers.GetUserByID(tiUserID)

* Fill the TreeView with the available roles and check
* off the ones the user is in.

loRoles = oSecurity.GetRolesForUser(tiUserID)
with This.pgfUsers.SFPage1.oRolesForUser
```

```
  .Nodes.Clear()
  for each loRole in oSecurity.oRoles
    loNode = .Nodes.Add(, 1, ;
      'R' + transform(loRole.ID), loRole.Name)
    loNode.Checked   = loRoles.GetKey(loRole.Name) > 0
    loNode.ForeColor = ;
      iif(loRole.ID = oSecurity.oRoles.iEveryoneID, ;
      rgb(172,168,153), rgb(0, 0, 0))
  next loRole
endwith

* Refresh the form.

This.Refresh()
```

Adding a user or a group

The Click method of the New button in the toolbar calls the Add method of the form, which handles adding either a user or a group, depending on which page is active. Add checks the lCanAdd property of the form; if it's .F., the user isn't allowed to add a record, so the method returns .F. lCanAdd is normally .T., but you could set it to .F. if you want to limit the number of users or groups available, perhaps for licensing reasons. Add also checks the lChanged property; if it's .T., something in the form changed, so Add calls the Save method to ensure changes to the existing record are saved before a new one is added. If Save returns .F. for some reason (for example, the user name was left blank), Add returns .F. so a new record cannot be added yet.

If the User page is the current one, Add calls the NewUser method of the oUsers member of the security object. NewUser returns an empty user object, which is based on Empty but with properties for the user attributes. Add puts this object into the oUser member of the form. Add then goes through the nodes in the roles TreeView on the Users page, setting each node's Checked property to .F. except for the Everyone node. This means new users are by default only members of the Everyone role. Add then sets focus to the user name textbox on the Users page.

For the Groups page, Add does almost the same thing, except it puts an empty role object in the oRoles member, unchecks all user nodes in the users TreeView (new roles have no users by default), and sets focus to the role name textbox on the Groups page. (Note that Add refreshes the textbox first, because it's disabled for certain roles, and we must re-enable it before setting focus to it.)

After creating a new user or role, Add then sets the lCanNavigate, lCanAdd, and lCanDelete properties to .F. and refreshes the form. This disables the New and Delete buttons and the users and roles listboxes so the user must save or cancel the new record prior to performing any other tasks. Our users find this is less confusing, but if you'd rather have all actions always available, simply comment out or delete the appropriate lines of code.

```
local llReturn, ;
  loNode, ;
  liRole
with This
  do case

* If we're not allowed to add a new item or the user
* changed the selected object and we can't save it,
* return .F.

    case not .lCanAdd or (.lChanged and not .Save())
      llReturn = .F.

* Create a new user.

    case .pgfUsers.ActivePage = 1
      .oUser = oSecurity.oUsers.NewUser()
      with .pgfUsers.SFPage1
        for each loNode in .oRolesForUser.Nodes
          liRole = val(substr(loNode.Key, 2))
          loNode.Checked = ;
            liRole = oSecurity.oRoles.iEveryoneID
        next loNode
        .txtUserName.SetFocus()
      endwith
      llReturn = .T.

* Create a new role.

    otherwise
      .oRole = oSecurity.oRoles.NewRole()
      with .pgfUsers.SFPage2
        for each loNode in .oUsersForRole.Nodes
          loNode.Checked = .F.
        next loNode
        .txtRoleName.Refresh()
        .txtRoleName.SetFocus()
      endwith
      llReturn = .T.
  endcase

* If we're creating a new object, disable navigation,
* adding, and deleting for now.

  if llReturn
    .lCanNavigate = .F.
    .lCanAdd      = .F.
    .lCanDelete   = .F.
    .Refresh()
  endif llReturn
endwith
return llReturn
```

Notice that when you add a user or role, the Cancel button is enabled but the Save button is only enabled once you've typed something in one of the textboxes. How is that done? If you remember my October 2005 article, "Raise a Little Event of Your Own," you know the answer. In case you missed it, here's a refresher. The InteractiveChange method of my textbox class (SFTextBox in SFCtrls.VCX) uses RAISEEVENT() to raise the AnyChange event. If the lBindToFormAnyChange property of the textbox is .T., which it is for all of the textboxes in this form, the Init method of the textbox uses BINDEVENT() to bind the AnyChange event of the textbox to the AnyChange method of the form. The AnyChange method of the form sets its lChanged property to .T. So, typing in any textbox automatically sets the lChanged property of the form to .T.

Similarly, the Init method of my commandbutton class (SFCommandButton in SFCtrls.VCX) uses BINDEVENT() to bind changes to the form's lChanged property to the OnFormChange method of the button if its lNotifyOnFormChange property is .T., which it is for the Save and Cancel buttons. OnFormChange calls the Refresh method of the button, which has code to enable the button when the lChanged property of the form is .T. (Cancel also checks lCanNavigate so it's enabled as soon as you add a user or role). So, typing in any textbox sets the lChanged property of the form to .T., which causes the Refresh methods of the Save and Cancel buttons to fire, which enable those buttons. So, there very little code in the form wiring up these events since that's all handled in my base classes. All I had to do was set the lBindToFormAnyChange property of the textboxes to .T., the lNotifyOnFormChange property of the Save and Cancel buttons to .T., and fill in the appropriate code for the Refresh method of the buttons (which I had to do anyway). The only wiring code is in the NodeCheck methods of the TreeView controls; they call Thisform.AnyChange so changes in the TreeView enable the buttons as well.

Saving or canceling changes

Whether you add a user or role or edit an attribute of an existing one, the Save and Cancel buttons are enabled as soon as you type something. The Click method of the Save button calls the Save method of the form. Save checks that the role name, user name, and password have been filled in (the password can be left blank if the lAllowBlankPassword property is .T.) and displays an appropriate error message if not. If everything is OK, Save calls the SaveItem method of the oUsers or oRoles member of the security object, depending on which page is selected, to save changes to the chosen user or role. Save then goes through the nodes in the appropriate TreeView, calling either AddUserToRole or RemoveUserFromRole, depending on whether a node is checked or not. In addition, AddUserToRole is called to ensure the chosen user is always in the Everyone role (AddUserToRole is smart enough to do nothing if the user is already in the specified role). Finally, Save calls the RestoreForm method, which restores the form to “non-edit” mode.

```
local llReturn, ;
  liUser, ;
  loNode, ;
  liRole
with This
  llReturn = .T.
  do case

* See if we have anything to do.

    case not .lChanged

* Ensure the role name was entered.

    case .pgfUsers.ActivePage = 2 and ;
      empty(.oRole.Name)
      .oMessage.ErrorMessage('ERR_BLANK_ROLENAME')
      .pgfUsers.SFPage2.txtRoleName.SetFocus()
      llReturn = .F.
    case .pgfUsers.ActivePage = 2

* Ensure the user name and password were entered.

    case empty(.oUser.UserName)
      .oMessage.ErrorMessage('ERR_BLANK_USERNAME')
      .pgfUsers.SFPage1.txtUserName.SetFocus()
      llReturn = .F.
    case not .lAllowBlankPassword and ;
      empty(.oUser.Password)
      .oMessage.ErrorMessage('ERR_BLANK_PASSWORD')
      .pgfUsers.SFPage1.txtPassword.SetFocus()
      llReturn = .F.
  endcase
  do case
    case not llReturn or not .lChanged

* If we're working with users, save the user and its
* roles. Ensure the user is in the Everyone role.

    case .pgfUsers.ActivePage = 1
      oSecurity.oUsers.SaveItem(.oUser)
      liUser = .oUser.ID
      with .pgfUsers.SFPage1.oRolesForUser
        for each loNode in .Nodes
          liRole = val(substr(loNode.Key, 2))
          if loNode.Checked
            oSecurity.AddUserToRole(liUser, liRole)
          else
            oSecurity.RemoveUserFromRole(liUser, ;
              liRole)
          endif loNode.Checked
        next loNode
      endwith
      oSecurity.AddUserToRole(liUser, ;
        oSecurity.oRoles.iEveryoneID)

* If we're working with roles, save the role and its
* users.

    otherwise
      oSecurity.oRoles.SaveItem(.oRole)
      liRole = .oRole.ID
      with .pgfUsers.SFPage2.oUsersForRole
        for each loNode in .Nodes
          liUser = val(substr(loNode.Key, 2))
          if loNode.Checked
            oSecurity.AddUserToRole(liUser, liRole)
          else
            oSecurity.RemoveUserFromRole(liUser, ;
              liRole)
          endif loNode.Checked
        next loNode
      endwith
  endcase

* Flag that we can navigate again, refresh the form,
* and requery the lists.

  if llReturn
    .RestoreForm()
  endif llReturn
endwith
return llReturn
```

The Cancel method, called from the Click method of the Cancel button, is very simple: it calls the RestoreForm method, the same method called by Save to reset the form. Here's the code for that method:

```
with This
  .lCanNavigate = .T.
  .lCanAdd      = .T.
  .lCanDelete   = .T.
  .lChanged     = .F.
  .RefreshForm()
endwith
```

Deleting users or roles

The Click method of the Delete button calls the Delete method of the form. This method simply calls the RemoveUser or RemoveRole method of the security object if you confirm the user or role should be deleted. Why not call the Remove method of the oUsers or oRoles members of the security object directly? While those methods would remove the specified user or role, they would leave orphaned records in the USERROLES and SECURITY tables. The RemoveUser and RemoveRole methods ensure all traces of the user or role are removed.

```
local llReturn
with This
  llReturn = .oMessage.YesNo('MSG_DELETE_RECORD')
  do case
    case not llReturn

* If we're working with users, remove the user.

    case .pgfUsers.ActivePage = 1
      oSecurity.RemoveUser(.oUser.ID)
      .RestoreForm()

* If we're working with roles, remove the role.

    otherwise
      oSecurity.RemoveRole(.oRole.ID)
      .RestoreForm()
  endcase
endwith
return llReturn
```

Summary

For the past three articles, I've discussed a framework for role-based security. We looked at classes that manage collections of users and roles, a security manager object responsible for all security-related tasks in an application, how to log users in and out, providing password encryption, and maintaining users and roles. The one piece we didn't look at is managing the rights roles have to secured elements in an application; that's left as an exercise for you.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com