Application Login Screen

What do we mean by application login? Because we are using single sign-on, we don't actually have the users enter a user name and password, but we do have them enter the two-factor authentication code. Our plan will be to remain on the Login screen until they have entered the correct two-factor code or they have exited the program. We need to handle several cases:

- When their account cannot do SSO, we need to tell them
- When they enter the wrong or an old two-factor authentication code, we need to tell them
- When they enter the correct code we need to continue

Behind the scenes, a lot is happening. Before the user has an opportunity to enter the two-factor authentication code, the Login screen will get the user identity from the OS and attempt to do SSO and then proxy using that OS user ID to the `apver` instance as the `appver` user. Remember the lengths we went to in order to obfuscate the `appver` password and the Java code that decodes it—all that is part of the process here.

Once a proxy session is established, the Oracle database determines if the user has entered a two-factor authentication code; if not, it generates and sends one to the user's mobile devices. To find out where to send the authentication code, Oracle database looks across a database link from the `apver` instance to the `orcl` instance, to the `HR` schema `EMPLOYEE,` and `emp mobile nos` tables to find out what devices are available for this specific user. Sending the authentication code to each device and returning to the Login screen are the next steps.

■ **Note** We are going to build this administrative interface for two Oracle database instances, `orcl` and `apver`. If you have not created the `apver` instance, you will need to use the code that is modified to run on a single instance, in the *Chapter12/single* directory.

Then, when the user receives the two-factor authentication code on their devices, and enters the code on the Login screen, a slew of additional steps are taken. The first step that the Login screen takes is to pass certain data and objects to the Oracle database. It passes the two-factor code, an inner class instance specific to this application, and an application identification string. Additionally, an RSA public/private key pair is generated on the client, and the modulus and exponent artifacts are also passed to the database.

Oracle database first assures that the two-factor authentication code the user entered matches what was sent to the user within the last ten minutes for this particular application, from this particular client network address. Then, if it is the correct code, the database assures the inner class instance matches what is stored for the application in question, and then retrieves the list of connection strings that is associated with it. Recall that the list of connection strings is stored in encrypted form on the database, and the code to encrypt and decrypt has been "wrapped" using the Oracle database *wrap* utility.

The Oracle database builds a RSA public key using the artifacts provided by the client, and generates a DES secret password key. Artifacts of the DES key are encrypted using the RSA public key to return to the client.

The list of connection strings for this specific application is encrypted with the secret password key, and that encrypted list is also returned to the client. At this point, the Login screen is about finished.

However, before it is closed and we continue on with the application, we use the artifacts of the DES secret password key to build a matching key on the client. We will use this key as long as we are in this particular application to decode connection strings for the application; so we clone or duplicate all the associated key components for continued use. Thereafter, the basic RSA and DES encryption key members are available for reuse by new keys associated with each Oracle connection we will be making for application transactions.

Everything I just described amounts to quite a list of functions, but it all occurs from the relatively simple interface presented in Figure 12-1. A single input field for the two-factor authentication code, and a Continue button are all we need to get the job done.



Figure 12-1. *Login screen*

The Application Inner Class

We are providing an application inner class structure in our `Login` class for use by any application. The code for the inner class is shown in Listing 12-1. By changing the package for the `Login` class, we can use this inner class for multiple applications, taking the burden off individual developers to include an inner class in their code. In the case of our security administrative interface application, the package is `orajavsec`. The complete inner class name will `orajavsec.Login$InnerRevLvlClass`. If there were another application that defined the `Login` class in a package named `mynewpkg`, then the inner class would be named `mynewpkg.Login$InnerRevLvlClass`. This difference is sufficient to distinguish different and separate applications and their associated application connection strings. This is true even if the definition of the inner class is identical except for that package name.

Listing 12-1. *The Login Class, Inner Class*

```
package orajavsec;

public class Login extends JDialog {

    public static class InnerRevLvlClass
                implements Serializable, RevLvlClassIntfc
        {
        private static final long serialVersionUID = 2011010100L;
        private String innerClassRevLvl = "20110101a";

        public String getRevLvl() {
            return innerClassRevLvl;
        }
```

```
    }
...
```

The code for Login is something we can give to other application programmers for inclusion in their applications. It is a template and a reusable piece of code—they just need to include it in their package. Then they don't need to worry about any of the setup details. After login is completed, they can just get their connection strings from `OracleJavaSecure` by calling the `getAAConString()` method.

In order to use the `Login$InnerRevLvlClass` in the Oracle database, we need to have a representative class to instantiate in the database. We need to execute the script shown in Listing 12-2 to create that Java structure in the database. We will be creating a similar Java structure in the database for each application. The only difference between classes for different applications will be the package name.

You may execute this script as the `sys` user on the `apver` instance, but you do not need to. When we get to the end of this chapter, we are going to use this security administration interface to register itself through a bootstrap process. That process will include creating the representative inner class in the Oracle Database and generating the connection strings that will be used by this application.

Listing 12-2. *Script to Build Login Inner Class*

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED APPSEC."orajavsec/Login" AS

package orajavsec;

import java.io.Serializable;

import orajavsec.RevLvlClassIntfc;

// Drop the "extends JDialog" from class definition
// It is unneeded and will be invalid on Oracle server
public class Login {
    public static class InnerRevLvlClass implements Serializable,
                                                     RevLvlClassIntfc {
        private static final long serialVersionUID = 2011010100L;
        private String innerClassRevLvl = "20110101a";

        public String getRevLvl() {
            return innerClassRevLvl;
        }
    }
}
/
```

■ **Note** This script can be found in the file *Chapter12/Login.sql*.

Center Method

We need to have the Login screen centered on the user's monitor, or else it may appear but be unnoticed in the top left corner of the monitor. Centering GUI interfaces is standard practice, and is something we

will use for all our user interface screens. Because we want to center all our GUI screens, we will place the method in `Login` , as shown in Listing 12-3, and make the method both public and static—anyone can call it without referring to an instance of `Login`. It is part of the template and is code we do not need to duplicate for each screen. We hand whatever item we want to have centered to this method, and the method adjusts the item's location.

Listing 12-3. GUI Center() Method

```
public static void center(Component item) {
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    Dimension frameSize = item.getSize();
    if (frameSize.height > screenSize.height) {
        frameSize.height = screenSize.height;
    }
    if (frameSize.width > screenSize.width) {
        frameSize.width = screenSize.width;
    }
    item.setLocation((screenSize.width - frameSize.width)/2,
                     (screenSize.height - frameSize.height)/2);
}
```

The code for the `center()` method is often generated automatically by the IDE for GUI applications (this version is from JDeveloper). I've just moved the code from a standard GUI `main()` method into this static method. Listing 12-3 shows the code.

JAVABEANS

I have mentioned JavaBeans, but this is our first introduction to them. JavaBeans are Java classes that have a required set of methods and interface. In particular, a JavaBean is a GUI object that has getter and setter methods for each property of the user interface. In this way, a JavaBean can be provided to an IDE, and the IDE automatically knows how to present it on screen and how to provide access to its properties. Each piece of text, input box and button that we have in our GUI application is a JavaBean.

Each IDE handles JavaBeans a little differently, placing the property settings code in a bit different location. Often the IDE will flag this code with a comment like, "This code is automatically generated, do not modify it." On the contrary, you generated the code when you drug or painted the component on the GUI application screen in the IDE, and you can edit the code. However, if you modify the code in unexpected ways, the IDE may no longer be able to present it to you on the screen. Perhaps a better wording for the flag comment would be, "This code lets the IDE display your components, please don't modify or add code that the IDE can't understand."

In JDeveloper, one of the free Java IDEs from Oracle (the other being NetBeans), when you develop GUI applications, most of the code for managing the user interface is placed in a method named `jbInit()`. Because we find all the parameter settings there initially, it is tempting to add our modifications to the UI in that method also. However, I have found that it is better to generate another method (what I will call `ojsInit()`) and to code our modifications to the UI in that method instead. We change the standard

constructor to call our method right after the call to `jbInit()`. No matter what IDE you are using, you will want to have a similar segregation of code.

Login Screen Constructors

In Listing 12-4, you can see how we call `ojsInit()` right after the call to `jbInit()` in the constructor. Additionally, we have provided a third constructor that takes a `Frame` class. We will call that constructor, rather than the default constructor, and provide a reference to our main application class, `parent`. Then when the login process is complete, we will use this reference to return to our application.

Listing 12-4. *Login Screen Constructors*

```
public class Login extends JDialog {

    public Login() {
        this(null, "", false);
    }

    public Login(Frame parent, String title, boolean modal) {
        super(parent, title, modal);
        try {
            jbInit();
            ojsInit();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public Login(Frame parent) {
        // For Oracle Java Secure, call this constructor
        this(parent, "two-factor Authentication Login", true);
    }

    public Login(Frame parent, String applicationID, String appClassName) {
        // This replacement constructor is used when managing a selected application
        super(parent, "two-factor Authentication Login", true);
        try {
            this.applicationID = applicationID;
            Class applicationClass = Class.forName(appClassName);
            appClass = applicationClass.newInstance();
            jbInit();
            ojsInit();
        } catch (Exception e) {
            System.out.println("Specified application class is not available");
            System.exit(0);
        }
    }
...
```

There is also a fourth constructor that bypasses (repeats code from) the other constructors. That fourth constructor takes an alternate application name and inner class name. When called, this

constructor uses reflection to instantiate a class of the type named, and will use that class in place of the inner class of `Login`. This will rarely be used, but is needed in our security administration interface. We have to become different applications in order to modify the associated connection strings. We will talk about that in detail later in this chapter.

The `JDialog` class, which `Login` is extending, has a constructor that takes three arguments. You can see our call to the `JDialog` constructor, `super()`. The last of those three arguments is a `boolean` that designates whether the `JDialog` is modal. We set the modal `boolean` to `true` for the Login screen. We want it to appear on top of whatever other screens are visible for the current application.

■ **Note** When calling other constructors, even super class constructors, those calls need to be the first line in the calling constructor. That is why we needed an independent fourth constructor. We needed to instantiate a new alternate inner class before we call the initialization methods. That could not have happened if we wanted to call our original constructors to do initialization. That call to the original constructors would have to be done first.

The "Wait While Processing" Modal Dialog

In a GUI application, bad things can happen if a user is impatient and your application code is off doing some complex calculation or data retrieval. The user may press a button repeatedly, possibly causing your application's code to do a function multiple times; or the user may quit the application in frustration. When your application is going to do something complex or possibly taking a long time, it is best to put up a little notification in front of the application, letting the user know you are busy working for them and asking them to be patient. Having described the complex list of tasks that take place in the Login screen, you can understand why it might take a few moments to process, during which time we will want the user to wait patiently.

I have defined a very simple `JDialog` class, `sayWaitDialog`, which I placed in the `Login` class. `sayWaitDialog` is a `static` member of `Login`, and it is configured by a `static` initializer block. This dialog screen is defined as a *modal* dialog—see the bold code in Listing 12-5. Modal dialogs appear on top of other windows on the user's monitor, and they can't easily be hidden behind other windows, at least those associated with the current application, even if the user clicks on other windows. I'm sure you have seen and noticed that kind of behavior in other dialogs before.

Listing 12-5. *Modal Dialog Asking User to Wait Patiently*

```
public static JDialog sayWaitDialog = new JDialog();
static {
    sayWaitDialog.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
    sayWaitDialog.setModal(true);
    sayWaitDialog.setTitle("Please Wait");
    JPanel jPanel1 = new JPanel();
    sayWaitDialog.setSize(new Dimension(255, 93));
    sayWaitDialog.getContentPane().setLayout(null);
    jPanel1.setBounds(new Rectangle(5, 5, 230, 45));
    jPanel1.setLayout(null);
    jPanel1.setBackground(new Color(255, 222, 214));
```

```
        JLabel jLabel1 = new JLabel();
        jLabel1.setText("Working. Please wait!");
        jLabel1.setBounds(new Rectangle(5, 5, 220, 35));
        jLabel1.setHorizontalAlignment(SwingConstants.CENTER);
        jLabel1.setFont(new Font("Tahoma", 0, 16));
        jPanel1.add(jLabel1, null);
        sayWaitDialog.getContentPane().add(jPanel1, null);
        Login.center(sayWaitDialog);
    }
```

Notice that the last line in the definition of `sayWaitDialog` is a call to our `center()` method to place the dialog in the center of the monitor. The last thing to mention about this dialog is in the first line. We set the dialog to do nothing on close. We disable the ability of the user to click on the X in the top right corner of the window to close the dialog. We actually never open or close the dialog; we simply make it visible when we want it and invisible when we don't.

Background Processing Thread

Now let's look at the `ojsInit()` method that we use to set additional properties of the Login screen GUI and to jump-start to login process. Listing 12-6 shows the code. In the bottom half of the method code, we see the call to set the static `Login.sayWaitDialog` to be visible and our call to center the Login screen and then make it visible. We also slightly modify the `Login Screen` by setting the `reEnterLabel` to be invisible. If the user enters an incorrect two-factor authentication code, we will set that help message to be visible.

Listing 12-6. *Self-Managed GUI Initialization and Background Processing Thread*

```
    private static String applicationID = "OJSADMIN";
    private static Object appClass = new InnerRevLvlClass();

    private void ojsInit() {

        SwingUtilities.invokeLater(new Runnable() {

                public void run() {
                    OracleJavaSecure.setAppContext(applicationID, appClass, "");
                    // Under some circumstances, this will throw an exception
                    // if OS User not allowed, also test isAppverOK
                    OracleJavaSecure.getAppConnections();
                    // on success, original error message will be blanked out
                    if (OracleJavaSecure.isAppverOK())
                        twoFactCodeTextField.setText("");
                    Login.sayWaitDialog.setVisible(false);
                }
            });
        Login.sayWaitDialog.setVisible(true);
        Login.center(this);
        reEnterLabel.setVisible(false);
        appIDLabel.setText(applicationID);
        this.setVisible(true);
        return;
    }
```

So what's that complex bit of code at the top of our `ojsInit()` method? There is a block of code that we want to run in the background while our modal `sayWaitDialog` is being displayed. That sounds easier than it is. You see, a Java GUI application like this is generally *single-threaded*—that is, only one track of code is processed at a time. This single focus is made even more stubborn by the fact that we are using a modal dialog. The modal dialog completely dominates the single-threaded processing until the dialog is removed (made invisible, in our case).

In older versions of Java, I have had to write separate `Thread` pools to handle out-of-band processing, but in current Java versions, the `SwingUtilities` class provides a standard approach to this problem. We hand a new `Runnable` instance (`Thread`) to the `SwingUtilities.invokeLater()` method (you can see this in Listing 12-6), and it delays running the `Thread` until your current code has processed a bit further. The delay seems to work well enough for our purposes.

You can see the new `Thread` that's created in the `new Runnable()` syntax. This is an anonymous class definition. In the class definition, we define a single method named `run()` that does our work. You can see that the `run()` method does the significant initial work of the `Login` class—everything up to waiting for the user to enter a two-factor authentication code. Most of that work is accomplished in the two calls to methods in `OracleJavaSecure`, `setAppContext()` and `getAppConnections()`.

The call to `getAppConnections()` is prone to failure under one specific condition that we need to address—whenever the OS user is not authorized to proxy connect through the Oracle application user. This can occur under some circumstances with an exception being thrown, and in other circumstances with the Oracle connection as `appver` user being null. We have added a method in `Oracle Java Secure` to test whether the `appver` connection is null: `isAppverOK()`. In both sets of circumstances, we need to inform the user. However, in the delayed `Thread`, it is difficult to catch the exception and do anything helpful with it. The situation here is one where we only need to notify the user of the fact. They need to log out and back in as a different user to use the application. Here's how we will tell them: the initial value we set in `jbInit()` for the `twoFactCodeTextField` is "Your OS User account cannot log in." If we succeed at making our proxy connection, then no `Exception` will be thrown, and we will continue on through our delayed `Thread`. This will set the `twoFactCodeTextField` equal to a blank string. However, if there is an exception we will not change that value, and the text field will remain as the original error message. Instead of calling this approach "error reporting," I would call this "lack of success reporting". Keep "lack of success reporting" in mind as a viable approach in cases like this.

At the end of our delayed `Thread`, `run()` method, we make the `sayWaitDialog` invisible. The code order in `ojsInit()` is opposite of the chronological order of execution. So we end up making the `sayWaitDialog` visible before we make it invisible.

The Continue Button

The last set of activities we want to address are those that occur after the user has entered a two-factor authentication code and either pressed the enter key or clicked on the Continue button. Both of those events call the `continueButton actionPerformed()` method. This method is shown in Listing 12-7. Notice that we call the same two methods of `OracleJavaSecure` that we called earlier; however, this time we provide the two-factor authentication code from `twoFactCodeTextField` that the user presumably received and entered.

Listing 12-7. *Continue with Provided two-factor Authentication*

```
private void continueButton actionPerformed(ActionEvent e) {
    if (twoFactCodeTextField.getText().equals("Bad two-factor code"))
        twoFactCodeTextField.setText("");
    if (twoFactCodeTextField.getText().equals("") ||
        twoFactCodeTextField.getText().equals("Your OS User account cannot log in"))
```

```
            return;
        OracleJavaSecure.setAppContext(applicationID, appClass,
            twoFactCodeTextField.getText());
        OracleJavaSecure.getAppConnections();
        if (!OracleJavaSecure.test2Factor()) {
            twoFactCodeTextField.setText("Bad two-factor code");
            reEnterLabel.setVisible(true);
            return;
        }
        this.setVisible(false);
        return;
    }
```

This time, after we return from the `getAppConnections()` method, we can test whether the two-factor code that was entered was successful by calling `OracleJavaSecure.test2Factor()`. This is a new method added in this chapter, the code for which is shown in Listing 12-8. We simply test the size of our list of connection strings that was returned as a result of providing our two-factor authentication code to the `getAppConnections()` method. A size of 0 is okay, and any other value is okay—it means the two-factor code was successful and `test2Factor()` returns a `true`. However, if the call to `consHash.size()` results in an `Exception` being thrown, then the list of connection strings is null, and we determine that the two-factor code was not acceptable, so we return a `false`.

Listing 12-8. *Test Success of two-factor Code from OracleJavaSecure*

```
    public static boolean test2Factor() {
        try {
            connsHash.size();
        } catch( Exception x ) {
            return false;
        }
        return true;
    }
```

Back in our `Login` class, in Listing 12-7, if the two-factor code is not good, we set our message in `twoFactCodeTextField` to "Bad two-factor code" and we make the `reEnterLabel` text visible. We are presenting user-feedback in the same text field used for user entry. Look back at the top of that method, `continueButton actionPerformed()`. You can see how we handle messages in the field where we want the user to enter the two-factor authentication code. If they have a blank or one of our error message strings in that field, we do not try to submit it as a two-factor authentication code, but simply return.

A successful login event, where the two-factor code is acceptable and a list of connection strings has been returned causes the last lines of `continueButton actionPerformed()` to be run. In that case, we basically set the Login screen to be invisible.

The Login Screen Closes

One final trick you might be interested in here: if the user exits the Login screen without completing the intended activities, then we exit the entire JVM that is running the application. This is handled through the `this WindowClosing()` method with a call to `System.exit()`, which is shown in Listing 12-9. The method, `this WindowClosing()` gets called automatically whenever the user clicks the X in the upper right-hand corner of the window to close it. In the normal process of entering an application, we will not close the `Login` screen, but will simply make it invisible.

Now you might be asking how the Login screen became visible in the first place, and what happens when it is no longer visible. The Login screen is called from an application, and control is returned to the application after the Login screen process, but I'm getting ahead of myself, and we will see this in the next section.

Listing 12-9. *Closing the Login and Exiting the Application*

```
private void this windowClosing(WindowEvent e) {
    System.exit(0);
}
```

Security Administration Menu

We will build an application that uses all the security features we've described in this book, and we will use the application to manage the security features—we will "eat our own dog food," so to speak. We can call this application `OracleJavaSecure` (OJS) administration and it will consist of a Menu, shown in Figure 12-2, and several Functional screens.



Figure 12-2. *Security administration menu*

The menu will reside in a class named `OJSAdmin`, and that class has a `main()` method that simply instantiates a new `OJSAdmin` class. See Listing 12-10. On instantiation, the `OJSAdmin` class runs through its JavaBeans initialization method, `jbInit()` and our additional initialization in the `ojsInit()` method.

Listing 12-10. *Security Admin Menu main() method and Constructor*

```
public static void main(String[] args) {
    // Put main() method right in this JFrame
    new OJSAdmin(args);
}

public OJSAdmin(String[] args) {
    try {
        jbInit();
        // Add extra init in separate method
        ojsInit(args);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The `OJSAdmin.ojsInit()` method is shown in Listing 12-11. Its primary functions are to instantiate a new `Login` class, which we discussed previously, and then center itself and make itself visible. Instantiating the `Login` class is all that is needed to use the security features of this book. Note in Listing 12-11 that we call one of the `Login` constructors if there are two (or more) arguments to `ojsInit()`, and another `Login` constructor if there are not. There will be arguments if we are calling the menu in order to act on behalf of a different application (discussed later).

Listing 12-11. *Additional OJSAdmin Menu Initialization*

```
private void ojsInit(String[] args) throws Exception {
    // Method for initializing OJS functionality
    JPanel disablePanel = bottomMenuPanel;
    // Login does SSO, two-factor Auth and Application Authorization

    if (args.length < 2)

        new Login(this);
    else {
        // Call Login with alternate Application ID and Class name
        new Login(this, args[0], args[1]);
        disablePanel = topMenuPanel;
    }
    // By default, we only use the top menu, so disable bottom components
    // When managing alternate application, we only use bottom menu
    Component[] comps = disablePanel.getComponents();
    for (int i = 0; i < comps.length; i++) {
        comps[i].setEnabled(false);
    }
    // This static utility method centers any component on the screen
    Login.center(this);
    // Finally, to see this frame, it must be made visible
    this.setVisible(true);
}
```

In the menu, we are going to differentiate two sets of menu buttons that we are going to display. These sets of buttons are located on two Java Swing `JPanel` components, `topMenuPanel` and `bottomMenuPanel`. Collecting the buttons in this way allows us to disable one of the sets of buttons by walking through all the components on a `JPanel` in a `for` loop and calling the `setEnabled()` method for each component.

We split the buttons into two sets because for functions represented by the `topMenuPanel` we will be acting with the privileges of the security administration interface (this application) and using Oracle connections associated with that application. However, for the functions represented by the `bottomMenuPanel` we will connect as a different, designated application and will be managing the connection strings associated with that application.

When the Login screen closes, after a successful two-factor authentication, the `OJSAdmin` screen becomes visible, because thread processing for the instantiation of `Login` has completed, and returns to the `OJSAdmin.ojsInit()` method, Listing 12-11. At that point, the application user can select from the menu options. If, for example, the user selects the **Add/Modify User** button, the event will call the `addUserButton actionPerformed()` method (Listing 12-12), which instantiates a JavaBean for that activity, `new AddUser()`. Each of those activity screens that are available from the `OJSAdmin` menu will both hide the `OJSAdmin` menu screen and make it visible once again. They can do this because we pass the menu by reference to the constructor for the activity screen—note the reference to `this` in the call to instantiate a `new AddUser` class in Listing 12-12.

Listing 12-12. *Action Method for the Add /Modify User Button*

```
private void addUserButton actionPerformed(ActionEvent e) {
    new AddUser(this);
}
```

Add/Modify User Functional Screen

Before we delve into the code and operation of our Add/Modify User functional screen, let's review the reasons we have this screen in the first place. Recall that we added a table in the `HR` schema named `t emp mobile nos`. We are using that table to store cell phone and pager numbers, and we also store (for convenience) the operating-system user ID of the employee. We need to maintain that data to accomplish both single sign-on and two-factor authentication. Additionally, we use the e-mail address in the `HR.EMPLOYEES` table for two-factor authentication, so we want to be able to edit that.

In order to maintain the `HR` data, we want a functional screen like our Add/Modify User screen, shown in Figure 12-3. This security administration interface is not the right place to maintain employee data, but we will permit it here in order to demonstrate our secure query and update procedures that we developed in Chapter 7. We will enable viewing and updating the sensitive `SALARY` and `COMMISSION PCT` columns.

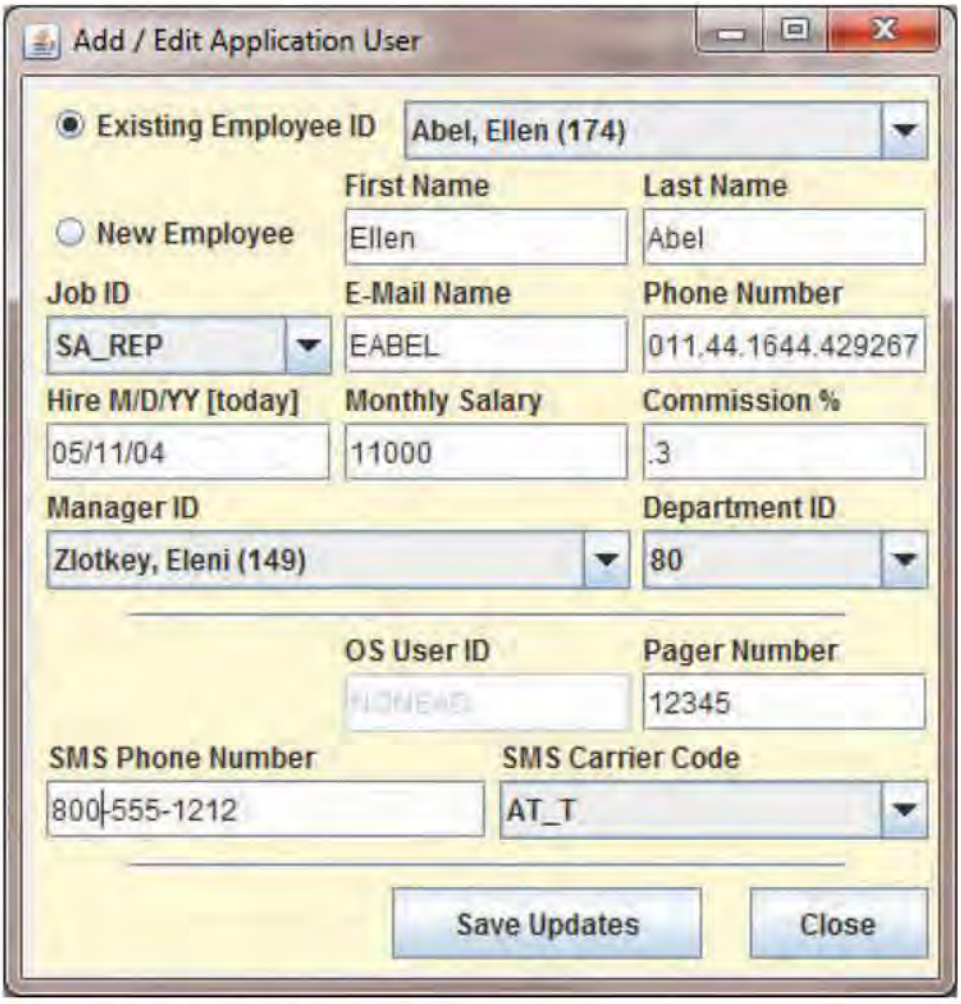


Figure 12-3. *Add/Modify User functional screen*

Instantiate the AddUser Screen

The Add/Modify User code is found in the file *Chapter12/orajavsec/AddUser.java*. We will begin exploring this functional screen by examining the constructors. Both constructors are shown in Listing 12-13. The first constructor is the one we call from the `OJSAdmin` menu. We pass a reference to `OJSAdmin`, so that we can hide it by calling the `parent.setVisible()` method. At about the same time, we set the Add/Modify User screen visible. Note that the first line in any constructor may be a call to a super-class constructor or another constructor in the same class. Here we call `this()`, which is a call to the default constructor (with no arguments); that is, the second constructor shown in Listing 12-13.

The last thing I'd like to point out about the first constructor shown in Listing 12-13 is that we end it by testing for the database connection being null. The connection would be null for at least a couple possible reasons, but the most important are that the operating system user account does not have access to the database. Specifically, the OS user may not have the right to proxy through the Oracle application user account that needs to be used for this function. When that occurs, we do not proceed; rather, we throw up a `Dialog` screen to notify the user. In this GUI single-threaded world, we will wait

until the user clicks on the button to acknowledge our message before we hide the Add/Modify User screen and make the OJSAdmin menu visible once again.

Listing 12-13. *Add/Modify User Screen Constructors*

```
public AddUser(JFrame parent) {
    this();
    this.parent = parent;
    // Post jbInit visual setup
    userMessageLabel.setVisible(false);
    userMessageLabel1.setVisible(false);
    ButtonGroup empGroup = new ButtonGroup();
    empGroup.add(existingEmpRadioButton);
    empGroup.add(newEmpRadioButton);
    existingEmpRadioButton.setSelected(true);
    Login.center(this);
    parent.setVisible(false);
    this.setVisible(true);
    if (null == conn) {
        JOptionPane.showMessageDialog(thisComponent,
            "Your account is not permitted to use this functional screen!");
        parent.setVisible(true);
        this.setVisible(false);
    }
}

public AddUser() {
    try {
        jbInit();
        conn = OracleJavaSecure.getAAConnRole("orcl", "appusr");
        // Possibly reentering - need new keys for new Oracle session
        OracleJavaSecure.resetKeys();
        locModulus = OracleJavaSecure.getLocRSAPubMod();
        locExponent = OracleJavaSecure.getLocRSAPubExp();
        sessionSecretDESPassPhrase = null;
        dataInit();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

In our second constructor shown in Listing 12-13 we do the following four things:

- Call `jbInit()` to initialize the GUI components.
- Call `getAAConnRole()` to get a connection, specific for this function. Because we are performing tasks related to the employee data in the `HR` schema, we are going to need the `hrview role` that we get through the `appusr` user on the `orcl` instance.
- Reset the encryption keys for this new connection.
- Call `dataInit()` to populate the drop-down combo boxes with existing values.

Probably the most important thing to know here is that we need to establish new encryption keys whenever we get a new connection, even if we are re-entering this same screen multiple times from the `OJSAdmin` menu. In order to assure we are cleaning up after ourselves, we call `closeConnection()` whenever we close this window. That happens in the `this WindowClosing()` method shown in Listing 12-14. We also see in Listing 12-14 what we described about the `AddUser` class making itself invisible and making the `OJSAdmin` menu screen visible again when done. Please use this approach as a template for opening and closing connections and resetting encryption keys.

Listing 12-14. *Window Closing Method*

```
private void this windowClosing(WindowEvent e) {
    OracleJavaSecure.closeConnection();
    parent.setVisible(true);
    this.setVisible(false);
}
```

In addition to calling `resetKeys()` in the default constructor for `AddUser`, we get a new RSA key pair, storing the exponent and modulus in static members. We also have static members in `AddUser` for the artifacts of the DES secret password key. We set one artifact, `sessionSecretDESPassPhrase` to be null in Listing 12-13. We test this static member later to see whether we need to re-acquire the secret password key. If we are doing an update or insert before we do a select, then we may need to call an additional stored procedure to exchange keys, `p get shared passphrase`.

Each of the functional screens in our application requires that we connect to a specific Oracle instance (`orcl` or `apver`) and that we connect as a specific user. The screens, instances, users, and roles are listed in Table 12-1. Different users are required because each screen requires different security privileges that we will acquire through different Oracle roles that are set based on the specific application and proxy user. The role is set through our `getAAConnRole()` method and the `p check role access` procedure. That procedure finds the relation between the application/proxy user and the required role by querying the `appsec.v application registry` view. We need to insert the relationship data shown in Listing 12-15 as the `appsec` user on the `orcl` instance.

Table 12-1. *OJSAdmin Functional Screen Users*

Screen Class	Instance	User	Role
AddUser	orcl	appusr	hrview_role
AdminUsers	orcl	ojsaadm	ojs_adm_admin
AssignApp	orcl	ojsaadm	ojs_adm_admin
RegNewApp	apver	avadmin	appver_admin
PickAppManage	apver	avadmin	appver_admin
EditAppConns	apver	appver	{none}
Copy2NewVer	apver	appver	{none}

Listing 12-15. *Script to Associate Application Users to Roles*

```
INSERT INTO appsec.v application registry
(application id, app user, app role) VALUES
('OJSADMIN','APPUSR','HRVIEW ROLE');

INSERT INTO appsec.v application registry
(application id, app user, app role) VALUES
('OJSADMIN','OJSAADM','OJS ADM ADMIN');

INSERT INTO appsec.v application registry
(application id, app user, app role) VALUES
('OJSADMIN','AVADMIN','APPVER ADMIN');
```

■ **Note** This script can be found in the file Chapter12/OrclAppSec.sql

During the call to the `Login` class associated with the `OJSAdmin` application in Listing 12-6, we designated `OJSADMIN` as the application ID. For that application ID and the specific application inner class, `ojsadmin.Login$InnerRevLvlClass`, there is (or will be) a list of connection strings stored in the `v app conn registry` view. It is this list that we got from the processing in `Login`. Using connection strings from the list, we can connect as the different users required for the various security administration functions. The current function, Add/Modify User, requires that we have a connection to `orcl` as `appusr` with the `hrview role`.

There is a lot of code in `AddUser` to make the GUI operate like it should, depending on user selections and entries. I will leave most of those details for you to explore and discover in the code, but I will mention a few important aspects. For instance, if the user is creating a new employee, they cannot enter data for the OS user identity until they have saved the employee and selected her from the drop-down combo box list. In the first constructor in Listing 12-13, we added a `ButtonGroup` and added our two radio buttons, `existingEmpRadioButton` and `newEmpRadioButton` as members. The button group assures that only one of the radio buttons is selected at a time. We also assure that personal data that was previously shown in the screen is blanked out when creating a new employee and/or user. Keeping track of what all is available and what becomes invalid as users click around on a GUI interface makes the programmer into a "bean counter" accountant.

Initialize the Data Selection Components

From the default constructor, shown second in Listing 12-13, after getting a new Oracle connection and resetting our encryption keys, we call a method to populate the drop-down combo box lists, `dataInit()`. The `dataInit()` method in Listing 12-16 has several static Oracle queries to non-sensitive views in the `HR` schema. Although the data is non-sensitive, we have not granted `PUBLIC` access to the data; rather, we granted access to the `hrview role`, which we acquire and use in this Add/Modify User functional screen. The `applicationID` is `OJSADMIN` and the application user is `appusr`, so the corresponding role that we acquire from `v application registry` is the `hrview role` according to Table 12-1 and Listing 12-15.

Using that Oracle role, we can select the list of employees from the public view, `hr.v employees public`. We created that view in Chapter 2, in the *Chapter2/HR.sql* script file. We order

the list by last name of the employee, and we concatenate the last name, a comma, the first name, and the `employee id` in parentheses. We do similar queries in `dataInit()` to populate the job ID, department ID, and manager drop-down combo lists. Notice in Listing 12-16 that, before we process the results through the `while(rs.next())` loop, we first remove all existing items in the list and we add a blank string as the first item. We may call `dataInit()` repeatedly, after every update, because the lists of values may have changed as a result of our update. So, we clear the old listings and add new listings during initialization. We add the blank string as our first item in each list, as the default selection.

Listing 12-16. *AddUser Data Initialization Method*

```
    private void dataInit() throws Exception {
        Statement stmt = null;
        ResultSet rs = null;
        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery("SELECT last name || ', ' || first name || " +
                "' (' || employee id || ')' " +
                "FROM hr.v_employees_public ORDER BY last name");
            // This throws event to run existingEmpComboBox actionPerformed() method
            // Calls blankAll()
            existingEmpComboBox.removeAllItems();
            existingEmpComboBox.addItem("");
            while (rs.next()) {
                existingEmpComboBox.addItem(rs.getString(1));
            }
            if (rs != null)
                rs.close();
...
```

The list in `existingEmpComboBox` becomes our source for determining which employee the application user wants to view or edit. When the application user selects an employee, we need to find the corresponding data in the `HR` schema, and we want to find that data by `employee id`. The plan will be to pull the employee ID out from between the parentheses of the selected item in `existingEmpComboBox`, so we will create a static utility method named `pullIDFromParens()`, Listing 12-17, that we will place in a class named Utility. We will use this in several places in this application, so we want it to be centrally located and not repeated as a member method of several functional screens.

Listing 12-17. *Get Data Index from Data Value*

```
    static String pullIDFromParens(String inValue) {
        String rtrnValue = "";
        try {
            int openPlace = inValue.indexOf("(");
            int closePlace = inValue.indexOf(")", openPlace);
            if (openPlace > -1 && closePlace > -1)
                rtrnValue = inValue.substring(openPlace + 1, closePlace);
        } catch (Exception x) {
        }
        return rtrnValue;
    }
```

We feel confident in using a static Oracle query instead of a stored procedure to get the list of employees and other lists in our `dataInit()` method (see Listing 12-16) because there are no application-user-provided parameters, so no chance of SQL injection and the data is non-sensitive, though still protected by a role grant (it's not `PUBLIC`). This doesn't go against what we've said about using stored procedures for queries; it is in addition to that approach. Again, querying the data directly from a view is safe as long as the query is static with no application-user-provided parameters.

In order for the `hrview role` to query data from the `HR.v employees public` view, we need to grant the `SELECT` privilege to `hrview role`. We make that grant as the `HR` user, along with a grant to select from `HR.v sms carrier host` in Listing 12-18. In `dataInit()`, we also populate the contents of the job ID, department ID, manager, and SMS carrier drop-down boxes, so we need to select from both these views.

Listing 12-18. *Grant Further Select on HR Views*

```
GRANT SELECT ON hr.v employees public TO hrview role;
GRANT SELECT ON hr.v sms carrier host TO hrview role;
```

■ **Note** The script for these grants is located in the file named *Chapter12/OrclHR.sql*.

Select an Existing Employee

After the `AddUser` screen is initialized, the application user may elect to enter data and add a new employee, or they may select an existing employee from the combo-box list. When an existing employee is selected, the `existingEmpComboBox actionPerformed()` method is called. We again want to request that the application user be patient, so we make the `sayWaitDialog` visible—see the bottom of Listing 12-19. We show the structure of the `existingEmpComboBox actionPerformed()` method in Listing 12-19, but the bulk of the `run()` method of our delayed `Thread` will be shown in later listings.

Listing 12-19. *Method for Selecting Existing Employee*

```
    private void existingEmpComboBox actionPerformed(ActionEvent e) {
        // When action from dataInit() at removeAllItems(), getItemCount() = 0
        if (0 == existingEmpComboBox.getItemCount() ||
            0 == existingEmpComboBox.getSelectedIndex()) {
            osUserIDTextField.setEnabled(false);
            blankAll();
            return;
        }

        employeeID = Integer.parseInt(Utility.pullIDFromParens(
            (String)existingEmpComboBox.getSelectedItem()));

        blankAll();
        SwingUtilities.invokeLater(new Runnable() {
                public void run() {

        // The bulk of the run() method has been removed from this Listing

                    Login.sayWaitDialog.setVisible(false);
```

```
            }
        });
    // It may take a while to get the user data, esp while we set up encryption
    // So ask the user to be patient while working
    Login.sayWaitDialog.setVisible(true);
}
```

That method, existingEmpComboBox actionPerformed() gets the employee ID from the selected item by using the Utility.pullIdFromParens() method. Before we display the modal sayWaitDialog that will dominate the current GUI Thread process until dismissed or made invisible, we create a new Runnable Thread and pass it to the SwingUtilities.invokeLater() method. This delayed Thread will populate the AddUser screen with data for the selected employee.

In our delayed thread run() method, we call two Oracle stored procedures, shown in abbreviated form in Listings 12-20 and 12-22. The first procedure is one of our standard encrypted data queries from Chapter 7, p select employee by id sens. That procedure accomplishes key exchange, if not already done, and selects data from the HR.EMPLOYEES table for the selected employee id. The salary and commission pct values are returned in encrypted form. We call OracleJavaSecure.getDecryptData() to decrypt the values and then populate each field on AddUsers with the value we retrieved.

If there is an error with our call to p select employee by id sens, we display an error message dialog, which is a modal dialog. If there is no error, then we get the components of the shared password (DES) key from the Oracle database. Our first call to OracleJavaSecure.getDecryptData() will have the side effect of building an equivalent DES key for use in the client.

Listing 12-20. *Abbreviated run() Method when Selecting Existing Employee, Part 1*

```
    public void run() {
        int errNo;
        String errMsg;
        OracleCallableStatement stmt = null;
        OracleResultSet rs = null;
        try {
            stmt =
                (OracleCallableStatement)conn.prepareCall(
                "CALL hr.hr_sec_pkg.p_select_employee_by_id_sens(?,?,?,?,?,?,?,?,?,?,?)");
...
            stmt.setInt(10, employeeID);
            stmt.executeUpdate();

            errNo = stmt.getInt(8);
            if (errNo != 0) {
                errMsg = stmt.getString(9);
                JOptionPane.showMessageDialog(thisComponent,
                   "Oracle error p select employee by id sens) " + errNo + ", " + errMsg);
            } else {
                sessionSecretDESSalt = stmt.getRAW(3);
...
                rs = (OracleResultSet)stmt.getCursor(7);
                // Should be only one record for this Employee ID
                if (rs.next()) {
                    firstNameTextField.setText(rs.getString(2));
...
                    // Our stored procedure passes Hire Date back as sql.Date
```

```
            // So process here to format
            java.sql.Date sDate = rs.getDate(6);
            Date hDate = new Date(sDate.getTime());
            hireDateTextField.setText(hDateFormat.format(hDate));
            jobIDComboBox.setSelectedItem(rs.getString(7));
            deptIDComboBox.setSelectedItem(rs.getString(11));
            // Find this user's manager id in parentheses of combo box
            for (int i = 0; i < managerIDComboBox.getItemCount(); i++) {
                if (rs.getString(10).equals(Utility.pullIDFromParens(
                    (String)managerIDComboBox.getItemAt(i))))
                {
                    managerIDComboBox.setSelectedIndex(i);
                    break;
                }
            }
            // Decrypt salary and commission pct using shared password key
            salaryTextField.setText(OracleJavaSecure.getDecryptData(rs.getRAW(8),
                sessionSecretDESPassPhrase, sessionSecretDESAlgorithm,
                sessionSecretDESSalt, sessionSecretDESIterationCount));

            commissionPctTextField.setText(

                OracleJavaSecure.getDecryptData(rs.getRAW(9),

                sessionSecretDESPassPhrase, sessionSecretDESAlgorithm,
                sessionSecretDESSalt, sessionSecretDESIterationCount));
        }
        if (rs != null) rs.close();
        if (stmt != null) stmt.close();
    }
```

You can see in Listing 12-20 how we populate the `AddUser` form with certain data elements. We simply set the text of text fields on the form. For the date field, we take pains to convert the `java.sql.Date` into a `java.util.Date`, and we format the date using our static `hDateFormat` member, which was defined like this:

```
static SimpleDateFormat hDateFormat = new SimpleDateFormat("MM/dd/yy");
```

For our drop-down combo boxes, e.g., `jobIDComboBox`, we select the item in the existing list that corresponds with the value from our query. In the particular case of the `managerIDComboBox`, we walk through all the items in the existing list and call the `pullIDFromParens()` method to get the manager's employee ID. If that matches the manager ID we got from our query, we set that manager as our selected item. Also, you can see, we set the value for the `salaryTextField` and `commissionPctTextField` to the value we get from decrypting the data we received from our query.

We also populate the user data on the bottom half of the `AddUser` screen with data from the user mobile numbers view, `hr.v emp mobile nos`. To do that, we call a new procedure in a package that we are defining in the `HR` schema, `hr pub pkg.p select emp mobile nos by id`. The script for creating this package is shown in Listing 12-21. This script should look very familiar to you at this point. Notice there are two procedures defined, the select procedure we already mentioned, and a procedure that we will use to update the data in `v emp mobile nos`, named `p update emp mobile nos`. We grant execute on this package to `hrview role`.

Listing 12-21. *Script to Create the HR Public Package*

```
CREATE OR REPLACE PACKAGE BODY hr.hr pub pkg IS

    PROCEDURE p_select_emp_mobile_nos_by_id(
        m employee id             emp mobile nos.employee id%TYPE,
        resultset out         OUT RESULTSET TYPE,
        m err no              OUT NUMBER,
        m err txt             OUT VARCHAR2)
    IS BEGIN
        m err no := 0;
        OPEN resultset out FOR SELECT
            user id, com pager no, sms phone no, sms carrier cd
        FROM v emp mobile nos
        WHERE employee id = m employee id;
    EXCEPTION
        WHEN OTHERS THEN
            m err no := SQLCODE;
            m err txt := SQLERRM;
            appsec.app sec pkg.p log error( m err no, m err txt,
                'HR p select emp mobile nos by id' );
    END p select emp mobile nos by id;

    PROCEDURE p_update_emp_mobile_nos(
        m employee id        emp mobile nos.employee id%TYPE,
        m user id            emp mobile nos.user id%TYPE,
        m com pager no       emp mobile nos.com pager no%TYPE,
        m sms phone no       emp mobile nos.sms phone no%TYPE,
        m sms carrier cd     emp mobile nos.sms carrier cd%TYPE,
        m err no         OUT NUMBER,
        m err txt        OUT VARCHAR2 )
    IS
        test emp ct      NUMBER(6);
    BEGIN
        -- Note: Use of this procedure assumes you have already done a select
        -- and that you are using the same Session Secret PassPhrase
        m err no := 0;
        SELECT COUNT(*) INTO test emp ct FROM v emp mobile nos WHERE
            employee id = m employee id;
        IF test emp ct = 0
        THEN
            INSERT INTO v emp mobile nos
                (employee id, user id, com pager no, sms phone no, sms carrier cd)
            VALUES
                (m employee id, m user id, m com pager no, m sms phone no,
                m sms carrier cd);
        ELSE
            UPDATE v emp mobile nos
            SET user id = m user id, com pager no = m com pager no,
                sms phone no = m sms phone no,
                sms carrier cd = m sms carrier cd
```

```
            WHERE employee id = m employee id;
        END IF;
    EXCEPTION
        WHEN OTHERS THEN
            m err no := SQLCODE;
            m err txt := SQLERRM;
            appsec.app sec pkg.p log error( m err no, m err txt,
                'HR p update emp mobile nos' );
    END p update emp mobile nos;

END hr pub pkg;
/

GRANT EXECUTE ON hr.hr pub pkg TO hrview role;
```

The process of selecting data from the new `hr pub pkg.p select emp mobile nos by id` procedure is very similar to the procedure we reviewed earlier for selecting data from the `v employees public` view. However, in the case of the data from `v emp mobile nos`, there is no sensitive data, so none of the columns is returned in encrypted form. This procedure also gets called from the delayed `Thread run()` method of the `existingEmpComboBox actionPerformed()` method of `AddUser`. That call is shown in Listing 12-22. The only unique thing here is that we disable the `osUserIDTextField` when we populate it from this query. We do that because we do not allow the user ID to change once set for an employee. Disabling this field allows the value to be presented, albeit in grayed-out form, but doesn't allow the data value to be edited. If this query does not find values in `v emp mobile nos` for the employee ID, then we enable the `osUserIDTextField`.

Listing 12-22. *Abbreviated run() Method when Selecting Existing Employee, Part 2*

```
    // Select from mobile nos where emp id
    stmt = (OracleCallableStatement)conn.prepareCall(
        "CALL hr.hr_pub_pkg.p_select_emp_mobile_nos_by_id(?,?,?,?)");
...
    stmt.setInt(1, employeeID);
...
        if (rs.next()) {
            // Will not let you change a user ID for an employee
            osUserIDTextField.setEnabled(false);
            osUserIDTextField.setText(rs.getString(1));
            pagerNumberTextField.setText(rs.getString(2));
            smsPhoneNumberTextField.setText(rs.getString(3));
            smsCarrierCodeComboBox.setSelectedItem(rs.getString(4));
        } else {
            osUserIDTextField.setEnabled(true);
        }
```

These two Oracle stored procedures (shown in Listings 12-20 and 12-22) represent dynamic queries where the query includes the requirement that `employee id = user selected employee id`. Dynamic queries like this are susceptible to SQL injection, unless they are encapsulated in Oracle stored procedures like these. This is a design for security concern that should be considered in every security code review.

Create a New Employee

We can create new employees from the Add/Modify User functional screen. In that case, a new employee ID will be assigned. Until the employee has been created, we do not permit the application user to enter a user ID. In this way it is a two-step process. Once the employee is created, we can then select him from `existingEmpComboBox` and enter user and mobile number data for him.

Save Data for the Employee

Once the application user has entered data that she would like to save for the selected or new employee, she will select the Save button. That action calls the `saveButton actionPerformed()` method. We show the structure of that method in Listing 12-23. In that listing, we have left out the bulk of the delayed thread `run()` method, which will be presented later. We test the values entered on the AddUser form to assure that values have been entered for all required fields. After we test them, we once again make the modal `sayWaitDialog` visible. In the meantime, we define another delayed Thread to do our work. At the end of the delayed thread `run()` method, we set the `sayWaitDialog` invisible. As I have described the use of the delayed `Thread` and the `sayWaitDialog` several times now, I will stop mentioning them; although, we will continue to use those features throughout this application.

Listing 12-23. *Save Employee and User Data*

```
    private void saveButton actionPerformed(ActionEvent e) {
        if (lastNameTextField.getText().equals("") ||
            jobIDComboBox.getSelectedIndex() == 0 ||
            eMailNameTextField.getText().equals("") ||
            managerIDComboBox.getSelectedIndex() == 0 ||
            deptIDComboBox.getSelectedIndex() == 0) {
            JOptionPane.showMessageDialog(thisComponent,
                "Must have values for Last Name, Job ID, E-Mail, Dept ID and Mgr ID!");
            return;
        }
        if (existingEmpComboBox.getSelectedIndex() > 0 &&
            osUserIDTextField.getText().equals("") &&
            (!(pagerNumberTextField.getText().equals("") &&
               smsPhoneNumberTextField.getText().equals("")))) {
            JOptionPane.showMessageDialog(thisComponent,
                "Must have value for User ID, else blank mobile nos!");
            return;
        }
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
...
                Login.sayWaitDialog.setVisible(false);
            }
        });
        // Ask the user to be patient while working
        Login.sayWaitDialog.setVisible(true);
    }
```

The delayed Thread that is defined in the `saveButton actionPerformed()` method does our Oracle update processing. In this case, the real processing occurs in two update Oracle stored procedures. The

first is our encrypted sensitive data update procedure, `p update employees sensitive`, from Chapter 7. The second is a new update procedure in the `hr public pkg` package for data in the `hr.v emp mobile nos` view, `p update emp mobile nos` shown in Listing 12-21.

In Listing 12-24, we see the code used to call the `p update employees sensitive` procedure. We pass the current employee ID to indicate which employee we are updating. Notice that we set parameters eight and nine to the encrypted values of the salary and commission percent fields. We also take the formatted date entered for hire date and parse it using the `hDateFormat.parse()` method into a `java.util.Date`, then convert it into a `java.sql.Date` to pass to the Oracle stored procedure for update. If no entry exists in the Hire Date field, we submit the current date to the procedure. One of the user-interface protections we enforce in `AddUser` is that the application user may not edit the hire date of an existing employee.

■ **Note** This process removes time significance from the hire date. We are doing that just for simplicity here.

Listing 12-24. *Update Employee Data*

```
    stmt = (OracleCallableStatement)conn.prepareCall(
    "CALL hr.hr sec pkg.p_update_employees_sensitive(?,?,?,?,?,?,?,?,?,?,?,?,?,?)");
    stmt.registerOutParameter(12, OracleTypes.NUMBER);
    stmt.registerOutParameter(13, OracleTypes.VARCHAR);
    stmt.setInt(1, employeeID);
...
    if (hireDateTextField.getText().equals("")) {
        stmt.setDate(6, new java.sql.Date((new Date()).getTime()));
    } else {
        Date hDate = hDateFormat.parse(hireDateTextField.getText());
        stmt.setDate(6, new java.sql.Date(hDate.getTime()));
    }
    stmt.setString(7, (String)jobIDComboBox.getSelectedItem());
    stmt.setRAW(8, OracleJavaSecure.getCryptData(salaryTextField.getText()));
    stmt.setRAW(9, OracleJavaSecure.getCryptData(commissionPctTextField.getText()));
...
    stmt.executeUpdate();
```

We only call the `p update emp mobile nos` procedure in Listing 12-25 if we have selected an existing employee and if the `osUserIDTextField` is not blank. We do not do any further data validation here, but you may want to.

■ **Note** Adding an entry in the `hr.v emp mobile nos` view does not equate to creating an Oracle user with the user ID we entered in the view. However, we will need to create the Oracle user in order for that user to have access to our applications. As a corollary situation, we may drop (delete) an Oracle user, but the entry in `hr.v amp mobile nos` will not be automatically deleted.

Listing 12-25. Update User and Mobile Numbers

```
if (existingEmpComboBox.getSelectedIndex() > 0 &&
    (!osUserIDTextField.getText().equals(""))) {
    stmt = (OracleCallableStatement)conn.prepareCall(
    "CALL hr.hr_pub_pkg.p_update_emp_mobile_nos(?,?,?,?,?,?,?)");
...
```

If we are creating a new employee without having selected an existing employee previously, then we would not have exchanged encryption keys yet. If that's the case, then the `sessionSecretDESPassPhrase` member (among others) will be null. We test `sessionSecretDESPassPhrase` for null, and if it is we call an additional Oracle stored procedure, `p_get_shared_passphrase`, from Chapter 7, to exchange keys before calling the data update procedures.

WHEN DOES THE ORACLE USER GET CREATED?

As of yet, we have not created the Oracle user corresponding to user ID we entered on the Add/Modify User functional screen. For each employee in the `HR.EMPLOYEES` table, we can create a record in `hr.v_emp_mobile_nos` that contains a user ID. The user ID is the same as the name of a Windows (operating system) account. We will create an Oracle user with that same name, and we will use the fact of having identical names to do single sign-on. The Oracle user will be granted access because the person has already logged in as an identical operating system user.

You may now wonder why we never ran a procedure to create the Oracle user from our Add/Modify User functional screen. It wasn't an oversight, but merely a delay until the last minute. We have no reason to create an Oracle user if he does not have access to any applications, so we will only create the user when we first grant him access to an application. That will happen in either the Admin Users or Assign Application functional screens, described next.

User Administration Screen

In the security administration interface application, each of our functional screens has many similarities to one another. We discussed many of those features already in looking at the Add/Modify User functional screen, and we won't repeat those details. However, there are plenty of new ideas introduced in each screen that will give us food for thought. The Admin Users functional screen is our current topic of discussion.

In the Admin Users screen shown in Figure 12-4, we have a drop-down combo box where we will find a list of all the users, `userComboBox`. This is not a list of all employees, but rather a list of all those employees with records in the `hr.v_emp_mobile_nos` view. For each user, we get the user ID from `hr.v_emp_mobile_nos` and the first and last name from `hr.v_employees_public`. We accomplish this in the `dataInit()` method of the `AdminUsers` class, and the process is identical to what we did in the `dataInit()` method of `AddUser`, as described previously.



Figure 12-4. *Admin Users functional screen*

Also on the Admin Users screen, you will see three check boxes: one for each administrative proxy user: `appusr`, `ojsaadm`, and `avadmin`. When a user is selected from the drop-down list, these checkboxes will be populated with checkmarks or blanks corresponding to the current proxy grants for the user. By selecting or deselecting these check boxes and clicking on the Save Updates button, the application user can grant and revoke those privileges from the selected user.

There is one more button on the Admin Users screen, the Revoke User Access button. The function of this button stops short of deleting the Oracle user account; however, it revokes all the application grants that the user has, including the grant to proxy through `appver`—so the user in effect can no longer use our applications.

Create the OJSAAdm User

In the Admin Users functional screen, we are connecting to the `orcl` instance as the `ojsaadm` user. This is one of the proxy users we listed in Table 12-1. We need to create that Oracle user, and we need to grant him access to the `HR` views needed to populate the `userComboBox`. We will do this on the `orcl` instance as the Security Administrator, `secadm` user. The commands we are going to use are shown in Listing 12-26. When you execute those commands, substitute the OS username of the person you want to have manage the lists of administrators for the placeholder `osadmin`, and give the new `ojsaadm` user a strong password.

Listing 12-26. *Create the OJSAAdm User*

```
GRANT create session role TO ojsaadm IDENTIFIED BY password;
CREATE ROLE ojs adm admin IDENTIFIED USING appsec.p check role access;
ALTER USER osadmin GRANT CONNECT THROUGH ojsaadm;

GRANT SELECT ON hr.v employees public TO ojs adm admin;
GRANT SELECT ON hr.v emp mobile nos TO ojs adm admin;

GRANT CREATE DATABASE LINK TO ojsaadm;
```

```
GRANT CREATE VIEW TO ojsaadm;
GRANT CREATE PROCEDURE TO ojsaadm;
```

■ **Note** These commands are listed in the file *Chapter12/OrclSecAdm.sql*.

In addition to granting our new user the privilege to select from the HR views to populate the users list, you can see in listing 12-26 that we are giving this user some additional privileges. We are granting ojsaadm the system privileges to create database links, views, and procedures. We are going to put this Oracle user account to work. In fact, we are enlisting this Oracle user to talk across a database link from the orcl to the apver instance to accomplish user administration on both sides. That being said, we need to create an identical user on the apver instance. Listing 12-27 shows the script we will use to accomplish that. As usual, we are connecting as the SYS user to accomplish most of our work on the apver instance.

Listing 12-27. *Create the OJSAAdm User on the apver Instance*

```
GRANT create session role TO ojsaadm IDENTIFIED BY password;
GRANT SELECT ON sys.proxy users TO ojsaadm;
```

■ **Note** The script that is partly shown in Listing 12-27 is included in the file *Chapter12/ApverSys.sql*.

From what we have described for the functions of the Admin Users screen, you might imagine that this user needs to read some unusual data from the Oracle database in order to accomplish his tasks. One thing ojsaadm needs to do is to read from the data dictionary view, sys.proxy users. It is from that view that we will observe the avenue of access from the Oracle user through whom our OS user is matched by SSO, to the application role which is aligned with the Oracle user acting as the proxy. By adding or removing the ability to proxy through one of our application users, the individual person user is able or not to access the application.

Not just any Oracle user can modify data in the sys.proxy users view. In Listing 12-27, you see that we granted SELECT on that view to the ojsaadm user on the apver instance. We are going to make the same grant to ojsaadm on the orcl instance. It is not sufficient to accomplish our plan to just have SELECT privilege on that view. To grant and revoke proxy privileges, we need the ALTER USER privilege. Additionally, as I described at the end of our discussion of the Add/Modify User screen, we need the ability to CREATE USER at the time we grant them their first application proxy. To do this, we will create a package in the SYS schema called usr role adm pkg with procedures that will create users and grant and revoke proxy privileges. On the orcl instance, the script to create this package is shown in Listing 12-28. An identical package will be created on the apver instance.

In the usr role adm pkg package definition in Listing 12-28, you can see five procedures are defined. There are two procedures required to create a user, p create user once and p create user many. We have split that process up into two procedures because the first, where we actually create the Oracle user, will experience an exception if the user already exists. The second procedure, p create user many, is also part of the process to create an Oracle user account for application access. In that procedure, we grant the create session role to the user and grant him the permission to proxy through the appver user. That proxy is required in order to do SSO and two-factor authentication. Even if an Oracle user

already exists and was created with a password, he will not be able to use our applications unless he is able to proxy through appver. Notice that when we create new users strictly for application access, we do not give them a password; rather, we direct that they are IDENTIFIED EXTERNALLY. Usually, that would mean that we use the OS or a naming service to authenticate the user, but in our case, we authenticate our one, big application user (appver) and allow this IDENTIFIED EXTERNALLY user to proxy through. No authentication for the individual person Oracle user ever occurs.

There is also a procedure for revoking user access to our applications, p drop user. This name is a bit of an overstatement, because we do not actually drop the Oracle user account. That account may exist for some reason other than to access our applications. It may even have an associated password and allow the user to use the Oracle database in her own schema. All we really want to accomplish is to keep the user from accessing our applications, and we can do that by simply revoking the privilege to proxy through the appver user. That is all the p drop user procedure does.

Listing 12-28. *Create the sys.usr_role_adm_pkg Package*

```
CREATE OR REPLACE PACKAGE BODY sys.usr role adm pkg IS

    PROCEDURE p_create_user_once( username sys.proxy users.client%TYPE )
    AS
        PRAGMA AUTONOMOUS TRANSACTION;
    BEGIN
        EXECUTE IMMEDIATE 'CREATE USER ' || username || ' IDENTIFIED EXTERNALLY';
        COMMIT;
    EXCEPTION
        WHEN OTHERS
        THEN
            appsec.app sec pkg.p log error( SQLCODE, SQLERRM,
                'sys.usr role adm pkg.p create user once for ' || username );
    END p create user once;

    PROCEDURE p_create_user_many( username sys.proxy users.client%TYPE )
    AS
        PRAGMA AUTONOMOUS TRANSACTION;
    BEGIN
        EXECUTE IMMEDIATE 'GRANT create session role TO ' || username;
        EXECUTE IMMEDIATE 'ALTER USER ' || username || ' GRANT CONNECT THROUGH appver';
        COMMIT;
    EXCEPTION
        WHEN OTHERS
        THEN
            appsec.app sec pkg.p log error( SQLCODE, SQLERRM,
                'sys.usr role adm pkg.p create user many for ' || username );
    END p create user many;

    PROCEDURE p_drop_user( username sys.proxy users.client%TYPE )
    AS
        PRAGMA AUTONOMOUS TRANSACTION;
    BEGIN
        EXECUTE IMMEDIATE 'ALTER USER ' || username || ' REVOKE CONNECT THROUGH appver';
        COMMIT;
    EXCEPTION
```

```
        WHEN OTHERS
        THEN
            appsec.app sec pkg.p log error( SQLCODE, SQLERRM,
                'sys.usr role adm pkg.p drop user for ' || username );
    END p drop user;

    PROCEDURE p_set_proxy_through(
        username sys.proxy users.client%TYPE,
        proxyname sys.proxy users.proxy%TYPE )
    AS
        PRAGMA AUTONOMOUS TRANSACTION;
    BEGIN
        EXECUTE IMMEDIATE 'ALTER USER ' || username ||
            ' GRANT CONNECT THROUGH ' || proxyname;
        COMMIT;
    EXCEPTION
        WHEN OTHERS
        THEN
            appsec.app sec pkg.p log error( SQLCODE, SQLERRM,
                'sys.usr role adm pkg.p set proxy through for ' ||
                username || '/' || proxyname );
    END p set proxy through;

    PROCEDURE p_drop_proxy_through(
        username sys.proxy users.client%TYPE,
        proxyname sys.proxy users.proxy%TYPE )
    AS
        PRAGMA AUTONOMOUS TRANSACTION;
    BEGIN
        EXECUTE IMMEDIATE 'ALTER USER ' || username ||
            ' REVOKE CONNECT THROUGH ' || proxyname;
        COMMIT;
    EXCEPTION
        WHEN OTHERS
        THEN
            appsec.app sec pkg.p log error( SQLCODE, SQLERRM,
                'sys.usr role adm pkg.p drop proxy through for ' ||
                username || '/' || proxyname );
    END p drop proxy through;

END usr role adm pkg;
/

GRANT EXECUTE ON sys.usr role adm pkg TO ojs adm admin;
```

In order to grant and revoke privilege to access specific applications, we need to execute commands to `ALTER USER` and either `GRANT` or `REVOKE CONNECT THROUGH` the specific application proxy user that is used for the application. Those steps are accomplished by the `p set proxy through` and `p drop proxy through` procedures.

In each of these procedures, we concatenate a SQL command that represents what we want to accomplish and call `EXECUTE IMMEDIATE`, which is extremely handy for executing dynamic data definition

language (DDL) statements like CREATE USER and ALTER USER. All of these procedures are defined with the modifier PRAGMA AUTONOMOUS TRANSACTION. That allows each procedure to commit its own transaction.

The question arises, "why didn't we just grant the system privileges to CREATE USER and ALTER USER to ojsaadm?" First, we would still want to wrap these commands in procedures to control the parameters and avoid SQL injection attacks. And second, we want to limit what ojsaadm can actually do. Notice that in these procedures, ojsaadm can only create users that are IDENTIFIED EXTERNALLY. Beyond that, he can only grant and revoke the privilege to proxy through other users—that is much more limited ability than the grant of ALTER USER provides.

I mentioned that an identical package is to be created on the apver instance. The script for that is in the file *Chapter12/ApverSys.sql*. There is one difference between the instances in this regard. On the orcl instance, we grant execute on usr role adm pkg to the ojs adm admin role; while on the apver instance, we grant it directly to the ojsaadm user. Now the challenge for us is in how we will execute the commands on both database instances at one time. We will do this across a database link.

Enable the OJSAAdm User Across a Database Link

Connect as the ojsaadm user to the orcl instance. While there, you can test your access to the sys.proxy users data dictionary view. That command is shown first in Listing 12-29. The second command in Listing 12-29 is the command to create the database link. Change the password placeholder to the password you used to create ojsaadm on the apver instance.

■ **Caution** Take a moment to assure that the password for ojsaadm on the apver instance in particular is very complex. We have opened another avenue of attack on our perimeter by creating another user on the apver instance. With a very strong password (for example 15 random characters, mixed case with numeric and special characters) we can shore up our defenses. This is a password that will be used in the database link, but rarely typed in. We would prefer to disable the password for ojsaadm or to create a login trigger that denies him access, but in order to use our database link, ojsaadm needs to be able to log in on the apver instance.

Listing 12-29. *Create Link as OJSAAdm User*

```
SELECT * FROM sys.proxy users;

-- Private database link to apver as ojsaadm
CREATE DATABASE LINK apver link
CONNECT TO ojsaadm
IDENTIFIED BY password
USING 'apver';

CREATE OR REPLACE VIEW ojsaadm.instance proxy users AS
SELECT 'APVER' INSTANCE, proxy, client FROM sys.proxy users@apver link
UNION SELECT 'ORCL' INSTANCE, proxy, client FROM sys.proxy users;

-- Test the link and view
SELECT * FROM ojsaadm.instance proxy users;
```

```
GRANT SELECT ON ojsaadm.instance proxy users TO ojs adm admin;

-- To let appsec read view across database link
GRANT SELECT ON ojsaadm.instance proxy users TO appsec;
```

The third command in Listing 12-29 creates a view that will bring together the results of a select on `sys.proxy users` from both the `orcl` and `apver` instances. This view is a `UNION` of the results of two queries, one from each instance, and includes a prefix column indicating the instance on which the proxy was observed. Test the new view by selecting from it—this will also be a test of the grant on `apver` for `ojsaadm` to select from `sys.proxy users`. We will grant select on this view to all users who proxy through `ojsaadm` and acquire the `ojs adm admin` role.

The last command in Listing 12-29 provides one more avenue of access. You see, sometimes we need to be able to see all the proxy users on the `orcl` instance from procedures running on the `apver` instance. On `apver`, the `appsec` user already has a link to the `orcl` instance. That link was set up initially so that application security, `appsec` user could do two-factor authentication by selecting from `hr.v emp mobile nos` on `orcl`. Here we are extending the functionality of the `appsec` private database link from `apver` to `orcl` by allowing it to select all the proxy users from our new view. This also saves us from having to grant `SELECT` on `sys.proxy users` to `appsec` on the `apver` instance; she will be selecting from `proxy users` on `apver` by selecting the view on `orcl` which selects from `proxy users` across a link to `apver`. If this sounds circular, that's because it is.

Well, as we have observed, selecting from `sys.proxy users` does not meet the goals of our plan to manage proxy grants. From the Admin Users screen, we want to be able to grant and revoke the privilege to proxy through application users on both the `orcl` and `apver` instances. We also want to do that from a single Oracle connection, if possible. Is it possible to do this across a database link? It certainly is. In Listing 12-30 we see the definition of a package for just this purpose, `ojsaadm.apver usr adm pkg`. We will create the package in the `ojsaadm` schema, and the package will have access through the private database link, `apver link` to execute procedures on the `apver` instance.

Please notice in Listing 12-30 that we are calling the procedures across the link with a parameterized, `EXECUTE IMMEDIATE` statement. Our PL/SQL style call is defined in the `m stmt` string. For the variable (e.g., user name) we don't concatenate it with the statement string; rather, we have a placeholder for it, `:1`. We call that statement string from an `EXECUTE IMMEDIATE` command, and pass the username in to fill the placeholder by specifying `USING username`. This is another form of parameterized statement that is also impervious to SQL Injection like stored procedures and prepared statements in Java, which we discussed in the section called "Avoid SQL Injection" in Chapter 7.

Listing 12-30. *Package Definition for Executing Procedures Across a Link*

```
CREATE OR REPLACE PACKAGE BODY ojsaadm.apver usr adm pkg IS

    PROCEDURE p_create_apver_user( username VARCHAR2 )
    AS
        m stmt VARCHAR2(100);
    BEGIN
        m stmt := 'BEGIN sys.usr role adm pkg.p_create_user_once@apver_link( :1 ); END;';
        EXECUTE IMMEDIATE m stmt USING username;
        m stmt := 'BEGIN sys.usr role adm pkg.p_create_user_many@apver_link( :1 ); END;';
        EXECUTE IMMEDIATE m stmt USING username;
    END p create apver user;

    PROCEDURE p_drop_apver_user( username VARCHAR2 )
```

```
    AS
        m stmt VARCHAR2(100);
    BEGIN
        m stmt := 'BEGIN sys.usr role adm pkg.p_drop_user@apver_link( :1 ); END;';
        EXECUTE IMMEDIATE m stmt USING username;
    END p drop apver user;

    PROCEDURE p_set_apver_proxy_through( username VARCHAR2, proxyname VARCHAR2 )
    AS
        m stmt VARCHAR2(100);
    BEGIN
        m stmt :=
        'BEGIN sys.usr role adm pkg.p_set_proxy_through@apver_link( :1, :2 ); END;';
        EXECUTE IMMEDIATE m stmt USING username, proxyname;
    END p set apver proxy through;

    PROCEDURE p_drop_apver_proxy_through( username VARCHAR2, proxyname VARCHAR2 )
    AS
        m stmt VARCHAR2(100);
    BEGIN
        m stmt :=
        'BEGIN sys.usr role adm pkg.p_drop_proxy_through@apver_link( :1, :2 ); END;';
        EXECUTE IMMEDIATE m stmt USING username, proxyname;
    END p drop apver proxy through;

END apver usr adm pkg;
/

-- Grant to role
GRANT EXECUTE ON ojsaadm.apver usr adm pkg TO ojs adm admin;
```

The procedures in the `ojsaadm.apver usr adm pkg` correspond to procedures in the `sys.usr role adm pkg`, with one exception. The procedure in this package, `p create apver user`, calls both the `p create user once` and `p create user many` procedures in `sys.usr role adm pkg`. No exception is thrown by the `sys.usr role adm pkg` procedures, so we will succeed in calling `p create user many` even if the user already exists and `p create user once` fails. Finally, we grant `EXECUTE` on this package to the `ojs adm admin` role.

Select an Existing User

We return now from examining the Oracle structures and grants needed to accomplish our plan to the Admin Users functional screen. We will look first at the actions that occur when the application user selects an existing user from the drop-down `userComboBox`. The `userComboBox actionPerformed()` method gets called. We first get the user ID from within the parentheses in the selected item by calling the `pullIDFromParens()` method like this:

```
userID = Utility.pullIDFromParens((String)userComboBox.getSelectedItem());
```

Next, we create a delayed thread whose `run()` method contains the query shown in Listing 12-31. Notice that we select all the rows of the `instance proxy users` view – this is a static query, not vulnerable to SQL injection. We walk through the `ResultSet` testing each row for a value of `client` that matches our selected user. Then we check if the value of `proxy` matches any of our administrative users, and we select

the corresponding check box on the Admin Users screen. Before we got to this point, in the `userComboBox actionPerformed()` method, we called the `blankAll()` method, which had deselected all of the check boxes. So at the end of this procedure, the only check boxes that are selected are those representing proxy privileges that are granted to the selected user.

Listing 12-31. *Query to get Administrative Proxy Grants for the Selected User*

```
    rs = stmt.executeQuery("SELECT INSTANCE, proxy, client " +
        "FROM ojsaadm.instance proxy users ");
    while (rs.next()) {
        if (rs.getString(3).equalsIgnoreCase(userID)) {
            if (rs.getString(2).equalsIgnoreCase("OJSAADM"))
                ojsaadmCheckBox.setSelected(true);
            if (rs.getString(2).equalsIgnoreCase("APPUSR"))
                appusrCheckBox.setSelected(true);
            if (rs.getString(2).equalsIgnoreCase("AVADMIN"))
                avadminCheckBox.setSelected(true);
        }
    }
```

Save Updates to the Administrative Privileges

When the application user has selected a user and made changes to the administrative privilege check boxes that are selected, he will need to save the changes. He does this by selecting the Save Updates button. When that button is selected, the `saveButton actionPerformed()` method is called. Once again, the selected `userID` is pulled from the selected item in the `userComboBox`. A delayed thread is created with the code in Listing 12-32 in its `run()` method.

This method is extensive because we are independently calling procedures to grant or revoke each privilege represented by the check boxes on the Admin Users screen. We could have combined several of these calls into a dynamic loop, but we wouldn't have saved much code, and the clarity would have been diminished. The first step we take is to try to create the user on both the `orcl` and `apver` instances. The user may already exist, but this is an easy enough step and assures the user both exists and has the privileges needed to run our applications. We call three Oracle procedures to create the user—two operate on the `orcl` instance (`sys.usr role adm pkg.p create user once` and `p create user many`), and one operates across a database link to the `apver` instance (`ojsaadm.apver usr adm pkg.p create apver user`). Each of these procedure calls has the identical syntax, so I'll only show the full syntax once. Some of the procedure calls take one parameter, the user ID; some take two parameters, the user ID and the proxy user.

Listing 12-32. *Saving Updates to Administrative Privileges, Abbreviated*

```
    stmt = (OracleCallableStatement)conn.prepareCall(
        "CALL sys.usr_role_adm_pkg.p_create_user_once(?)");
    stmt.setString(1, userID);
    stmt.executeUpdate();
    if (stmt != null) stmt.close();


    stmt = (OracleCallableStatement)conn.prepareCall(
        "CALL sys.usr role adm pkg.p_create_user_many(?)");
...
```

```
    stmt = (OracleCallableStatement)conn.prepareCall(
        "CALL ojsaadm.apver usr adm pkg.p_create_apver_user(?)");
...
    // Next, grant or revoke each proxy
    if (ojsaadmCheckBox.isSelected()) {
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL sys.usr role adm pkg.p_set_proxy_through(?,?)");
        stmt.setString(1, userID);
        stmt.setString(2, "OJSAADM");
...
    } else {
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL sys.usr role adm pkg.p_drop_proxy_through(?,?)");
...
    }
    if (appusrCheckBox.isSelected()) {
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL sys.usr role adm pkg.p_set_proxy_through(?,?)");
        stmt.setString(1, userID);
        stmt.setString(2, "APPUSR");
...
    } else {
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL sys.usr role adm pkg.p_drop_proxy_through(?,?)");
...
    }
    if (avadminCheckBox.isSelected()) {
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL sys.usr role adm pkg.p_set_proxy_through(?,?)");
        stmt.setString(1, userID);
        stmt.setString(2, "AVADMIN");
...
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL ojsaadm.apver usr adm pkg.p_set_apver_proxy_through(?,?)");
...
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL ojsaadm.apver usr adm pkg.p_grant_apver_appver_conns(?)");
        stmt.setString(1, userID);
...
    } else {
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL sys.usr role adm pkg.p_drop_proxy_through(?,?)");
...
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL ojsaadm.apver usr adm pkg.p_drop_apver_proxy_through(?,?)");
...
        stmt = (OracleCallableStatement)conn.prepareCall(
            "CALL ojsaadm.apver usr adm pkg.p_revoke_apver_appver_conns(?)");
...
    }
    blankAll();
```

We call either `p set proxy through` or `p drop proxy through` for each of the administrative proxy users, depending on whether the check box is selected for that administrative proxy or not. In addition, for the proxy user avadmin, we also call the procedures that go across the link to grant or revoke proxy, `ojsaadm.apver usr adm pkg.p set apver proxy through` or `p set apver proxy through`, because this proxy user is used on both instances.

If the `avadminCheckBox` is selected we run one more procedure, `p grant apver appver conns`, and if not selected, we run `p revoke apver appver conns`. We will discuss those procedures in detail when we get to the Register New Application screen. The procedures grant or revoke another role we will need at that point.

Revoke User Access to Run Applications

When the administrative user selects a user on the Admin Users screen and then selects the Revoke User Access button, the `revokeUserButton actionPerformed()` method is called. The code that gets run in that method drops all the proxies for the selected user on both instances. This code is shown in Listing 12-33.

We instantiate an ordinary `Statement` class, `stmt2`, which will support a query that returns a `ResultSet`. This is in addition to the `OracleCallableStatement` that we instantiate to call our stored procedures. Similarly to the way we selected from `ojsaadm.instance proxy users` to set the administrative proxy grant check boxes when an existing user is selected from our `userComboBox`; here again, we select all the proxy users and only concern ourselves, as we walk through the `ResultSet`, with those proxy grants that were made to the selected user. Then if the instance where we saw the proxy grant is equal to "apver," we call the stored procedure to revoke the proxy grant across the database link. Otherwise, we call the stored procedure to revoke the proxy grant locally.

Listing 12-33. *Dropping all Administrative and Application Proxy Privileges*

```
stmt2 = conn.createStatement();
rs = stmt2.executeQuery("SELECT INSTANCE, proxy, client " +
    "FROM ojsaadm.instance_proxy_users ");
while (rs.next()) {
    if (rs.getString(3).equalsIgnoreCase(userID)) {
        if (rs.getString(1).equalsIgnoreCase("apver")) {
            stmt = (OracleCallableStatement)conn.prepareCall(
                "CALL ojsaadm.apver usr adm pkg.p_drop_apver_proxy_through(?,?)");
        } else {
            stmt = (OracleCallableStatement)conn.prepareCall(
                "CALL sys.usr role adm pkg.p_drop_proxy_through(?,?)");
        }
        stmt.setString(1, userID);
        stmt.setString(2, rs.getString(1));
        stmt.executeUpdate();
        if (stmt != null) stmt.close();
    }
}
if (rs != null) rs.close();
blankAll();
```

I considered writing this method a bit differently, only dropping those proxies that we know are related to our applications, and you may want to rewrite this (calling both `p drop user` and `p drop apver user` procedures to revoke the proxy grant to `appver`) if that is your goal. This method as written will drop all application proxies, including the privilege to proxy through the `appver` user, which

will keep the Oracle user from accomplishing SSO or two-factor authentication. We do not, however, drop the Oracle user account.

Application Assignment Screen

The Assign Application functional screen shares a lot of the concepts and code that we used in the Admin Users screen. Here, though, we are not dealing with the administrative proxy grants, but with grants to proxy for other applications. Figure 12-5 displays the Assign Application screen.

■ **Note** Users who are able to proxy through `ojsadmin` are able to assign any application to any users. Perhaps another level of filtering would be appropriate, but that has not been added in this book. On the positive side, these `ojsadmin` administrators can fill in for one another, assigning user access to each other's applications. The only application-specific administrative filtering we do here is in limiting which administrators can change connection strings (passwords) for which applications.

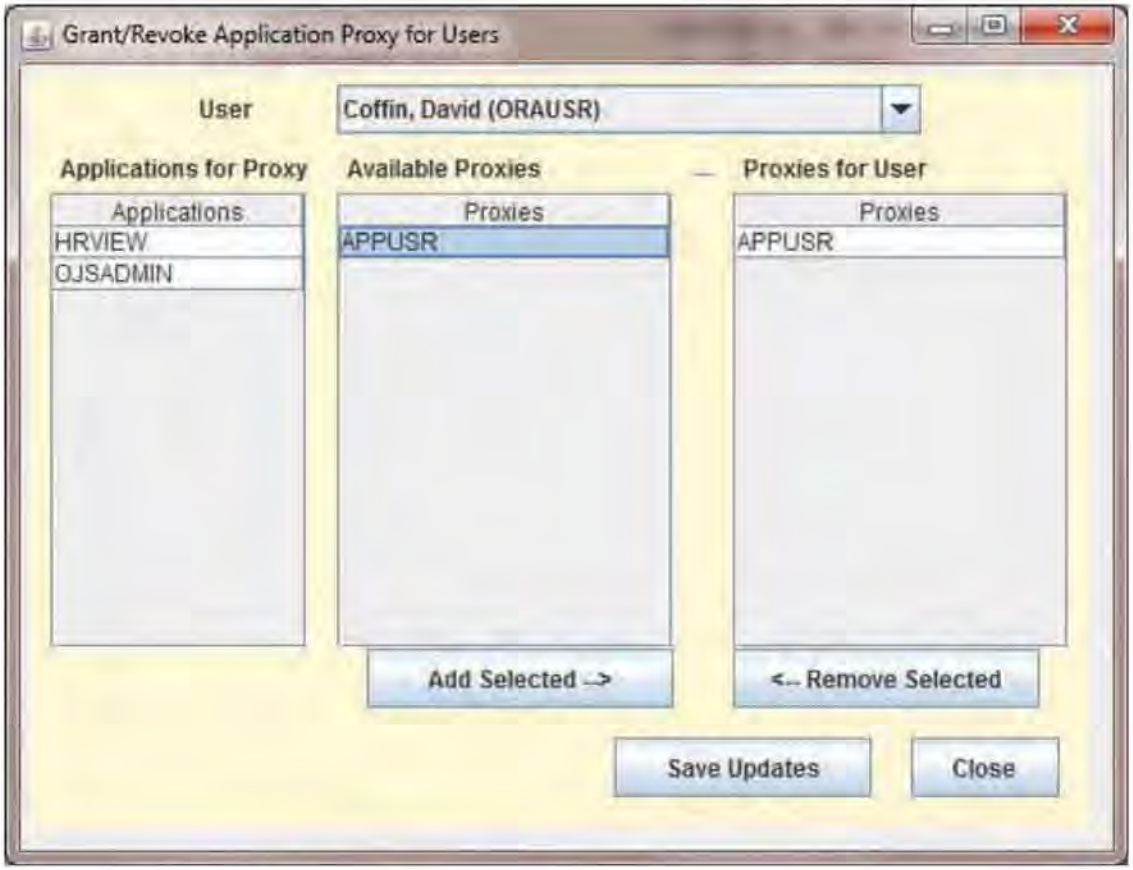


Figure 12-5. Assign Application functional screen

When there are many application proxies, having them all represented in a selectable table is easier than having a check box for each proxy. The one drawback in selecting from a list like this is the somewhat more likely possibility that an incorrect selection will be made and overlooked. For that reason, we don't show the administrative proxy users in this list.

The drop-down combo box at the top of the Assign Applications screen is where we select a user. We will be managing that user's access to applications. The far right table displays the list of proxy grants already given to the selected user, and the middle table lists all the available proxy grants. When an available proxy is selected, the far left table shows a list of applications in which that proxy is used. For example, the `appusr` proxy is assigned to users of the `HRVIEW` application as well as the `OJSADMIN` application.

These tables are a couple layers deep in nested GUI objects. Our tables reside in a scroll pane so that when the table grows beyond what can be seen in the little box we've provided, we will have vertical and/or horizontal scroll bars to give us access to the remaining data.

To add a proxy for the selected user, the Add Selected button will be pressed, and, likewise, to revoke a proxy from the selected user, the Remove Selected button will be pressed. To save these new settings, the application user will press the Save Updates button.

Initializing the Data Selection Components

When we instantiate a new `AssignApp` class, we will call the `dataInit()` method to populate three data-selection components. We will build the list of users for our `userComboBox`, as we have seen in other functional screens. We will also build a list of available application proxy accounts. And we will build a lookup table of lists of applications where a proxy user is used. Then, when a proxy user is selected from the available proxies list, we will display the associated list of applications in the far left column of the Assign Applications screen.

In our list of available proxy users for applications, we will not provide a true list of available proxies, because every Oracle user is in fact potentially a proxy-through user. We do not want to display all the Oracle users, or incorrect selections are bound to happen. Rather, the proxy users we will show in the available proxies list will be only those `app user` entries from the `appsec.v application registry` view that are not also security administration proxies. Recall that we grant and revoke access to the administrative proxy users through the Admin Users screen. The available proxies will be the list of `app user` entries for all registered, non-administrative applications.

For our available proxies list, we are going to populate a new GUI widget called a `JTable`. It is not so straightforward to populate the `JTable` as it was for our combo boxes. We cannot just call `addItem()`. In a `JTable`, there is an underlying table model that we populate. We are using instances of the `DefaultTableModel` that we declare as static members of `AssignApps`. This is shown in Listing 12-34.

Each table model has both data and column labels. The data and column labels can be provided to the table model in a variety of forms, and we elect to provide both the data and column labels as `Vectors`. We will only have a single column in our tables, which will be either a list of "Proxies" or a list of "Applications." Our column identifier vectors are defined as static members of `AssignApp` as shown in Listing 12-34. If we had multiple columns in our tables, we would add a label for each column to our column identifier `Vectors`. We do not have to tell the `JTable` how many columns of data we are going to have—it divines that from the number of entries in our column identifiers `Vector`.

Listing 12-34. *Static Members of Assign Applications*

```
static DefaultTableModel availableProxiesTM = new DefaultTableModel();
static DefaultTableModel userProxiesTM = new DefaultTableModel();
static DefaultTableModel appsTM = new DefaultTableModel();
static Vector columnIdentifiers = new Vector();
```

```
static {
    columnIdentifiers.add("Proxies");
}
static Vector appColumnIdentifiers = new Vector();
static {
    appColumnIdentifiers.add("Applications");
}
static Hashtable<String, Vector> appsHashtable =
    new Hashtable<String, Vector>();
```

The last static member we declare in Listing 12-34 is a `Hashtable`. The `appsHashTable` will have a key, the proxy user name, and a value, the `Vector` of applications where that proxy user is used.

A `DefaultTableModel` assumes that each row may have multiple columns, so it does not handle data as a simple list, even when there is only one column. Rather, in the case where `Vectors` are used, it expects each row to be a `Vector` of data elements, one for each column. So in the end, the data we provide to the table model in order to build our table will be a `Vector` of `Vectors` holding a list of potentially multi-column rows.

Now that you know where we are headed in building the data `Vectors` for our tables, let's look at the code from `dataInit()` that builds our available proxies list. This code is shown in Listing12-35. Look closely at the `while` loop where we handle the `ResultSet` from our query. We instantiate a new `itemVector` and populate it with one element, the first column of our `ResultSet`. The `itemVector` represents a row in our table. Then we add the `itemVector` to our `dataVector`. The `dataVector` represents a list of rows for the table. The last line in Listing 12-35 is a call to the available proxies table model `setDataVector()` method that will populate the table.

Listing 12-35. *Build Data for Available Proxies Table*

```
rs = stmt.executeQuery(
            "SELECT DISTINCT a.app user FROM appsec.v application registry a " +
    "WHERE a.app user NOT IN ('APPVER','AVADMIN','APPSEC','OJSAADM') " +
    "ORDER BY a.app user");
// dataVector must be Vector of Vectors
Vector dataVector = new Vector();
Vector itemVector;
while (rs.next()) {
    itemVector = new Vector();
    itemVector.add(rs.getString(1));
    dataVector.add(itemVector);
}
if (rs != null)
    rs.close();
availableProxiesTM.setDataVector(dataVector, columnIdentifiers);
```

In the `dataInit()` method we also build the data for our applications for proxy table. The contents of this table will change to match whatever proxy user is currently selected. For each proxy user, we will have a `Vector` of `Vectors` to use to populate the table. Those `Vectors` of `Vectors` will be held as values in `appsHashTable` that we can select with the key of the proxy user name.

Listing 12-36 shows the code used to build `appsHashTable`. In building the data, we again skip administrative proxies that are handled by the Admin Users screen, and we skip proxies that are used on the apver instance. We are prepared to call `dataInit()` multiple times, whenever the data may have changed, i.e., after the Save Updates button is selected. Because we may be rebuilding the `appsHashTable`, we start out by removing any existing content by calling the

clear() method. We also set the table model, appsTM to a null Vector—we don't want to show any applications until an available proxy is selected.

As we walk through the ResultSet, we check to see if we have already started a dataVector for the proxy user, rs.getString(2). We check if the appsHashTable contains a key by that name. If so, we get the value from appsHashTable which is our Vector of applications for the proxy, else we instantiate a new Vector. Again, each application is an item in a new Vector that is added to the dataVector for this proxy. Then the dataVector is put into the appsHashTable with the proxy user name as the key.

Listing 12-36. *Build Data for Applications for Proxy Table*

```
rs = stmt.executeQuery("SELECT DISTINCT a.application id, p.proxy " +
    "FROM ojsaadm.instance proxy users p, appsec.v application registry a " +
    "WHERE p.instance <> 'APVER' " +
    "AND p.proxy NOT IN ('APPVER','AVADMIN','APPSEC','OJSAADM') " +
    "AND a.app user = p.proxy ORDER BY a.application id");
// appsHashtable must be Hashtable of Vectors of Vectors
// empty static Hashtable each time you enter this screen
appsHashtable.clear();
appsTM.setDataVector(null, appColumnIdentifiers);
while (rs.next()) {
    if (appsHashtable.containsKey(rs.getString(2))) {
        dataVector = appsHashtable.get(rs.getString(2));
        itemVector = new Vector();
        itemVector.add(rs.getString(1));
        dataVector.add(itemVector);
    } else {
        dataVector = new Vector();
        itemVector = new Vector();
        itemVector.add(rs.getString(1));
        dataVector.add(itemVector);
    }
    appsHashtable.put(rs.getString(2), dataVector);
}
```

To build the table data for both the available proxies table and the applications for proxy table, we are selecting from the v.application registry view, and in this functional screen, we are connected as the ojsaadm user with the ojs adm admin role. In order to select from the appsec schema view, the proxy user needs SELECT privilege. As the Security Administrator, secadm on the orcl instance, we execute this grant:

```
GRANT SELECT ON appsec.v application registry TO ojs adm admin;
```

Selecting an Available Proxy in the Table

When the application user selects an available proxy from the table, the availableProxiesTable mouseClicked() method gets called. The significant code of that method is shown in Listing 12-37. To find the value of the data selected in a table, we need to get the value in a specified row, at a specified column. In a single-column table, all the data is in column index zero. So to get the value of the available proxy that was selected in the table, we get the value at the selected row and column zero. We use that selected proxy user name as the key to get the related dataVector from appsHashtable. Then we set the applications table model to use that dataVector.