14 Securing Fusion Web Applications

Security is an important part of any enterprise application. Security implementation in an application decides who can access the application and what they can do once they are logged in. This chapter describes how you can visually enable security in the different layers of your Fusion web application. The following topics are discussed in this chapter:

- Securing Fusion web applications
- Securing the business service layer
- Securing the view and model layer

Introduction

As you know, Oracle ADF is an end-to-end application framework built on top of the Java EE stack. The **Java Authentication and Authorization Services** (**JAAS**) is a Java security framework typically used for securing Java EE applications. Oracle fusion middleware simplifies the security model offered by Java and Java EE stack and provides a portable, feature-rich enterprise level security solution through **Oracle Platform Security Services** (**OPSS**).

[The OPSS is set of portable security services built on top of Java and Java EE stack abstracting the underlying security and identity management details. To learn more about OPSS, refer to the topic *Oracle Platform Security Services* available online at: `http://www.oracle.com/technetwork/middleware/id-mgmt/index-100381.html`.]

Oracle ADF security leverages the OPSS services for authentication and authorization of business users of the system. The ADF security framework at a high level offers the following:

- Integration with JDeveloper IDE to provide visual and declarative development
- Simplified security EL expressions which can be used in the web pages to control the access of UI elements
- End-to-end coverage and security of all layers of the application

The rest of the sections in this chapter will make you familiar you with the techniques you can use to implement ADF security for your Fusion web application.

Securing Fusion web applications

The JDeveloper IDE provides visual and declarative support to secure your ADF web application. It abstracts the underlying security implementation and frees you from the complexity of security APIs. In this section you will learn to use the JDeveloper design time features for securing a typical Fusion web application.

Configuring ADF security

In this section you will learn to enable authentication and authorization for your Fusion web application. To enable security for a Fusion web application, perform the following steps:

1. In the main menu, select the **Application | Security | Configure ADF Security** menu item. While doing so, the IDE will display the **Configure ADF Security** dialog.
2. In the first screen of the **Configure ADF Security** dialog, choose **ADF Authentication and Authorization** and click on **Next**.
3. Select the web project which needs to be secured and then **Authentication Type** in the next screen. The authentication types that you choose decide the login form type for your application.

The following are the authentication types displayed in this page:

- **HTTP Basic Authentication**: This uses the default browser login dialog.
- **HTTP Digest Authentication**: When you use this option the browser client encrypts the password before sending it to the server.
- **HTTPS Client Authentication** (public key certificate): This uses public key certificates to perform client authentication over a secure HTTP connection.
- **Form-Based Client Authentication**: When you select this option, the IDE allows you to choose either of the following:

    i. Choose to use the default login and error pages generated by the IDE.

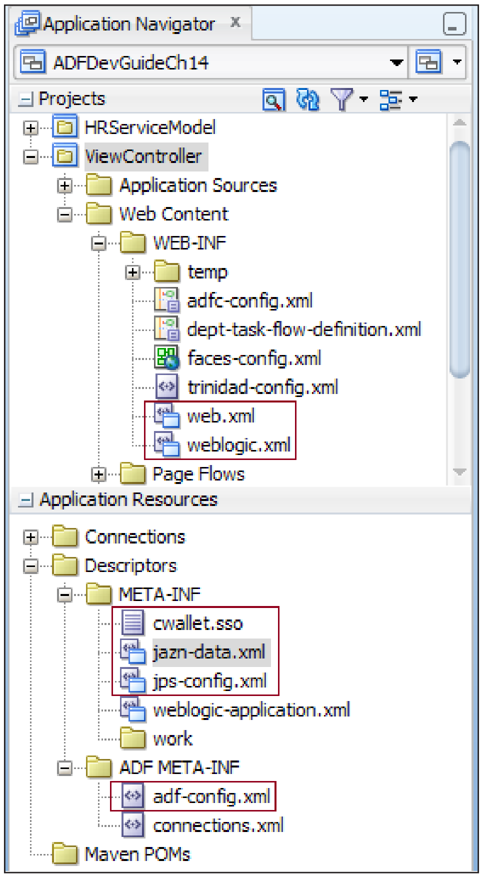
    ii. Choose the custom login and error page option, which will allow you to specify the custom login page you made.

For this example, select the **HTTP Basic Authentication** type and click on **Next** to continue.

4. Select **No Automatic Grants** in the **Automatic Policy Grant** screen. The **No Automatic Grants** option locks down all the application resources unless you explicitly specify access to them. Click on **Next**.
5. In the authenticated **Welcome** screen, choose **Redirect Upon Successful Authentication** and then configure the welcome page. If you specify the ADF Faces page, then make sure you added `/faces` at the beginning of the URL. For example, `/faces/welcome.jspx`.
6. Click on **Finish** to complete the wizard.

ADF security artefacts

When you enable security for a Fusion application, the IDE updates (or creates them if they not exist) the security related configuration files for the application, such as `web.xml`, `adf-config.xml`, `weblogic.xml`, `jps-config.xml`, `cwallet.sso`, and `jazn-data.xml`. Let us take a quick look these files and understand their role in managing security.



- **web.xml**: When you enable security, the IDE adds the following security related configurations in this file:

    i. **Authentication servlet for enforcing secured access to the system** (`oracle.adf.share.security.authentication.AuthenticationServlet`): When a request reaches the server, this servlet intercepts the request and checks whether the user has logged in to the system. If user is not logged in, then the system challenges the user with a login form.

    ii. **The JPS filter configuration** (`oracle.security.jps.ee.http.JpsFilter`): This filter is used for setting up the OPSS policy provider.

- **adf-config.xml**: This file stores the `JaasSecurityContext` entry which contains flags for enforcing the authentication and authorization checks.
- **weblogic.xml**: This file contains mapping between the valid users, security role and the OPSS user principal.
- **jps-config.xml**: This file contains metadata definitions for security services such as login modules, authentication providers, authorization policy providers, and credential stores.
- **jazn-data.xml**: This file contains the identity store and policy stores that you add while securing a Fusion web application.

  The identity store includes business users added to the application, and policy stores include resource types, permissions, application roles, and policy details describing what roles can access which resources. The identities are what authentication requests are done against.
- **cwallet.sso**: This file encrypts and stores the user credentials used in the application.

Once you enable security for an application, the steps that follow in securing the application are to define application roles, define users, role-user mapping, and granting appropriate resource permissions for roles. These are explained in the coming sections.

Defining application roles and users

To define application roles, perform the following steps:

1. Select **Application | Security | Application Roles** in the main menu. On selecting this menu item, the IDE opens up `jazn-data.xml` in the overview editor with the **Application Roles** tab selected.
2. To define new application roles, click on the green plus icon in the **Roles** panel and select the **Add New Roles** option from the drop-down menu. Enter **Name**, **Display Name**, and **Description** for the new role. In the **Mappings** tab, click on the green plus icon to add users and enterprise roles to the newly created role.

Note that you can group application roles under the existing roles to simulate the role hierarchy in an enterprise.

Application roles versus enterprise roles

When you define roles, you have two options. You can define them either as application roles or as enterprise roles. **Application roles** are local to an application and it can contain only users and roles defined in the application, whereas **enterprise roles** are available to all applications deployed in the domain. You will use these roles later while granting resource permissions.

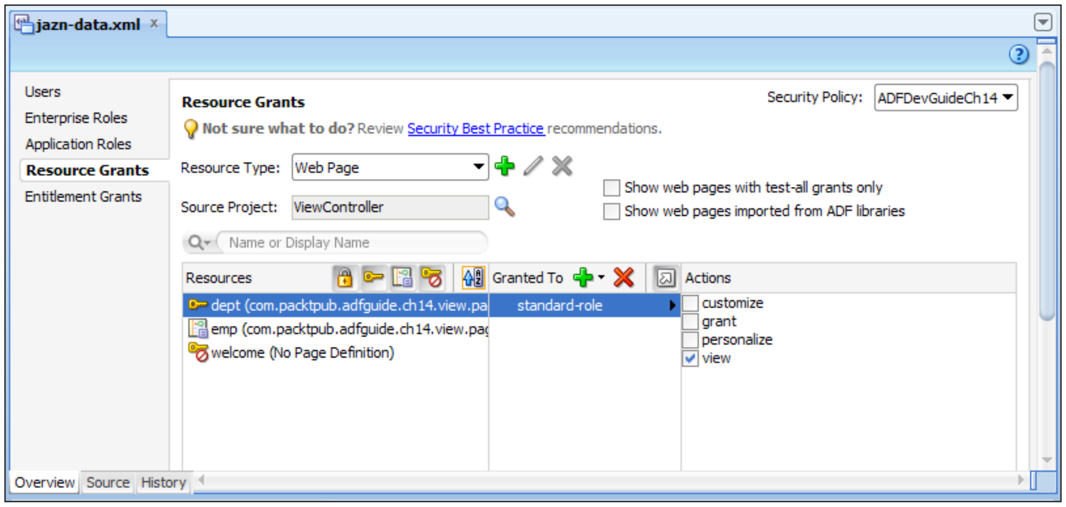
Although ADF security allows you to grant resource permissions to users through enterprise roles and application roles, as a best practice, is recommended to grant permissions directly to application roles alone in real-life applications. Later you can map users and enterprise roles to appropriate application roles. This approach gives you more flexibility in terms of security implementation.

Providing resource grants

In the previous sections we learned how to enable security in a Fusion web application and also learned about application roles. In this section we will learn how to configure access rights for each role. Access rights are defined using security policies which define roles or users who can access a resource, along with the actions they can perform on it.

To provide resource grants in an ADF application, perform the following steps:

1. Select the **Resource Grants** tab in the overview editor of `jazn-data.xml`.
2. Select the appropriate **Resource Type** in the drop-down list. The resource types listed are as follows: **ADF Entity Object**, **ADF Entity Object Attribute**, **ADF Method**, **Task Flow**, **Web Page**, and **Custom Resource Types**. You can use the search icon in the **Source Project** field to pick up the appropriate source project to look up the resources that need to be secured. For example, if you choose a web page as a resource type, the **Resources** table will list all the pages from the web project that you selected as the source. Note that you can secure only web pages with a page definition file in the **Resource Grant** screen.
3. You can use the **Resource Grant** screen to restrict the access of resources. To specify grants, select the appropriate resource in the **Resources** table and then click on the green plus icon in the **Granted To** panel header to add appropriate grantees, such as application role, user, enterprise role, and code source.
4. Once you have defined a grantee, the next step would be to specify the authorized actions that the grantee can perform on the resource. The following screenshot displays the **Resource Grants** editor window for the department page. In this example, the view action on the dept page is granted to all users belonging to **standard-role**.



Resource Grants

Security Policy: ADFDevGuideCh14

Not sure what to do? Review Security Best Practice recommendations.

Resource Type: Web Page

Source Project: ViewController

Show web pages with test-all grants only

Show web pages imported from ADF libraries

Resources	Granted To	Actions
dept (com.packtpub.adfguide.ch14.view.pa	standard-role	customize
emp (com.packtpub.adfguide.ch14.view.pag		grant
welcome (No Page Definition)		personalize
		✓ view

You must select the view action alone while granting actions on a web page. The other actions listed in the editor for a web page such as **customize**, **grant**, **view**, and **personalize** actions are for use by the Oracle Composer.

Using entitlement grants to aggregate resources

The entitlement grant allows you to group resources and corresponding actions into a single named security group that can be granted to application roles using a single grant statement. This simplifies the security configuration. For example, if your application has many task flows and web pages that are accessible to different application roles with similar action permissions, then you can create a single entitlement group with the appropriate resource grants and allowed actions on the granted resources. Later you can map the desired application roles to this group in one go.

To define the entitlement grant, perform the following steps:

1. Select the **Entitlement Grants** tab in the editor. Click on the green plus icon to add the entitlement. Specify **Name** and **Display Name**.
2. To add member resources to this security group, click on the green plus icon in the **Member Resources** panel header. Choose the desired resources in the **Select Resources** pop up. Select the appropriate actions for each member resource that you have added.

3. To map application roles to the grantee list for an entitlement, open the **Grants** tab and add application roles as appropriate.
4. You can click on the **Save** icon displayed in the main toolbar to save all changes that you made.

Securing the business service layer

The Oracle ADF security framework is very comprehensive and covers all layers of the application. In this section you will see the features offered by ADF Business Components for securing business data and business service methods.

Securing data update operations

ADF entity objects handle the posting of changes to the data source in a Fusion web application. You can leverage the security features offered by entity objects to authorize all the data update operations on entity rows. Oracle ADF allows you to authorize operations on an entity object at two levels:

- **Entity object level**: The entity level security settings are used to authorize operations such as read, update, and remove on entity object rows
- **Attribute level**: The attribute level security settings are used to authorize update on entity object attributes

To enable security for an entity object, open the desired entity object in the overview editor. Select the **General** tab and expand the **Security** section. The **Security** section displays **read**, **update**, and **removeCurrentRow** operations for the entity object. Select the appropriate operations that need to be enabled on the entity object.

To enable security at the attribute level of an entity object, select the desired attribute in the overview editor of the entity object. Switch to the **Security** tab and choose the **update** operation.

The following example illustrates the permission entry added for the `read`, `update`, and `removeCurrentRow` operations on the **DepartmentEO** entity object:

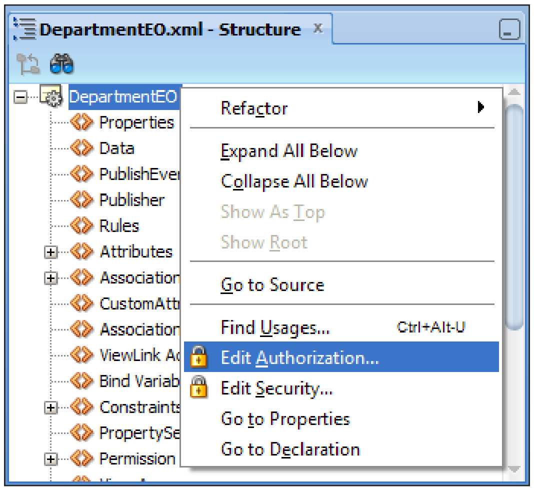
```
<Entity ...
 <Permission    target="com.packtpub.adfguide.ch14.model.entity.
DepartmentEO"     permissionClass="oracle.adf.share.security.
authorization.EntityPermission">
    <privilege-map
      operation="read"
      privilege="read"/>
    <privilege-map
      operation="update"
      privilege="update"/>
    <privilege-map
      operation="removeCurrentRow"
      privilege="delete"/>
  </Permission>
</Entity>
```

Once you have enabled the operations on an entity object, you can grant entity permissions to application roles.

To grant entity permissions to application roles, perform the following steps:

1. Select the appropriate entity object in the **Application** panel.
2. Go to the structure window, right-click on the security enabled entity object or entity attribute node, and choose **Edit Authorization** in the menu.

   This is shown in the following screenshot:



3. When you choose the **Edit Authorization** option, the IDE will display the **jazn-data.xml** file in the overview editor with the **Resource Grants** tab opened. Click on the **Add Grant** button displayed in the **Granted To** column header and select the appropriate grantees. Select the appropriate **Actions** that you want to grant.
4. Click on the **Save** icon in the main toolbar to save changes.

What happens at runtime?

At runtime when the user queries a view object that uses security-enabled entity objects or when the user updates a security-enabled entity object row, the framework will read the associated permission clauses for the underlying entity objects and perform a security check before carrying out the requested action. Based on the resource grant settings in the entity object, the framework will allow or disallow the operation.

Defining custom resource types

Oracle ADF security provides many features out of the box for securing various application resources. However, some enterprises may have specific authorization policies which may call for special treatment while enabling security for business applications. While working in such applications you may end up going beyond the built-in features and creating custom resource permissions for authorizing access to various application resources. In this section you will see how the ADF framework help you define custom permissions and use it to ensure authorized access of resources.

To define a custom resource type, perform the following steps:

1. Locate the **jazn-data.xml** file under the **META-INF** folder in the **Application Resources** panel.
2. Open the `jazn-data.xml` in the overview editor and select the **Resource Grants** tab. Click on the green plus icon located next to the **Resource Type** drop-down list.
3. In the **Create Resource Type** dialog, enter **Name**, **Display Name**, and **Description** for the custom resource type. The IDE will pre-populate the **Matcher Class** name to the `oracle.security.jps.ResourcePermission` class. This is a generic permission class that represents the access rights to resources of a particular type and allows the underlying policy providers to make authorization decisions on the resources as appropriate.

4. Click on the green plus icon displayed in the **Actions** panel (displayed at the bottom of the **Create Resource Type** dialog) to add appropriate actions for the resource type. Click on **OK** to save changes and dispose of the dialog.

   Once you have defined the resource type, you can map the resources to it by performing the following steps.

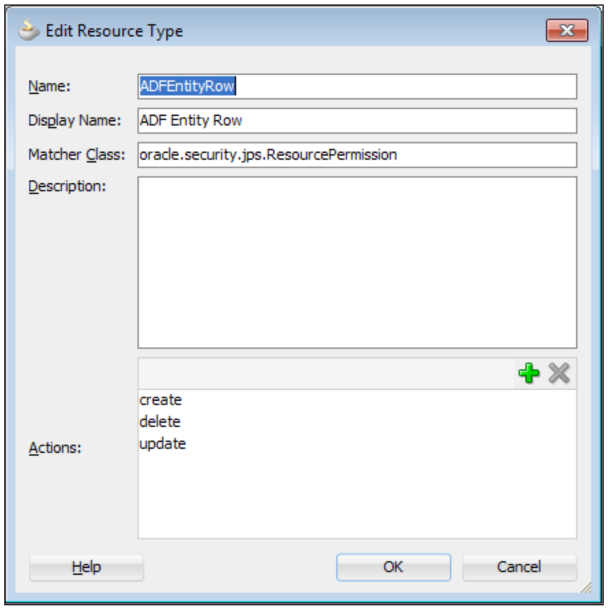
   1. Click on the green plus icon in the **Resource** panel window to map the resources to the resource type. In the **Create Resource** dialog choose **Resource Type**; specify **Name**, **Display Name**, and **Description**.
   2. Click on **OK** to save changes.

You can use the **Resource Grants** tab in the overview editor for granting access of resources types to desired roles. The following section is an example.

An example using a custom resource type for controlling data updates

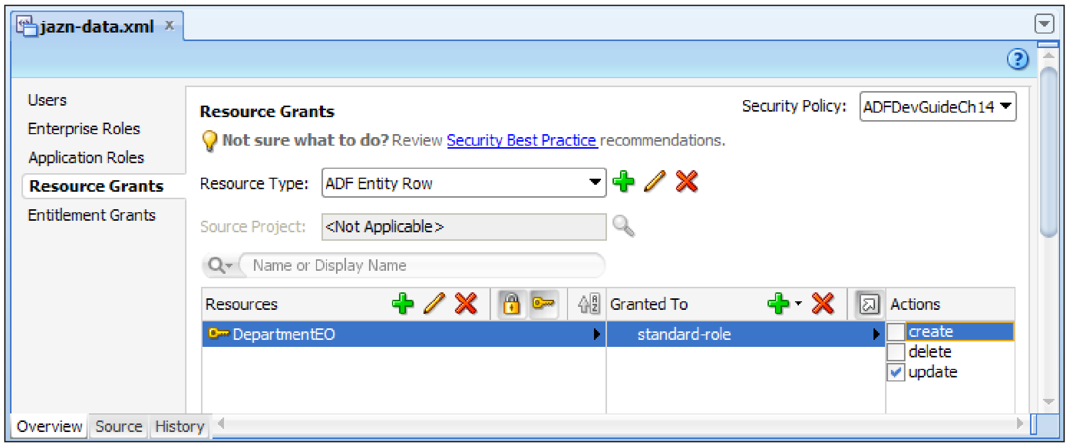
In this example we will see how a custom resource type can be used for controlling updates on an entity row. This example allows updates only on newly created entity rows.

The following screenshot displays the definition for the **ADFEntityRow** resource type used in the example. **ADFEntityRow** supports create, delete, and update actions which are displayed in the **Actions** section in the dialog:



Name: ADFEntityRow
Display Name: ADF Entity Row
Matcher Class: oracle.security.jps.ResourcePermission
Description:
Actions:
create
delete
update

This example uses **ADFEntityRow** to control operations on the **DepartmentEO** entity object. **DepartmentEO** is used in this example just to keep things simple; you are free to use any entity object name that you want to secure. To complete the security settings for this example you must add **DepartmentEO** as a resource to the **ADFEntityRow** type and grant appropriate resource permissions to suitable roles. The following screenshot displays the mapping of **DepartmentEO** to the **ADFEntityRow** type with the **update** action granted to a standard role.



Resource Grants

Security Policy: ADFDevGuideCh14

Not sure what to do? Review Security Best Practice recommendations.

Resource Type: ADF Entity Row

Source Project: <Not Applicable>

Name or Display Name

Resources	Granted To	Actions
DepartmentEO	standard-role	☐ create ☐ delete ☑ update

The basic infrastructural setup for the purpose of using a custom resource type for checking access rights should be ready by now. The following part will show you how to use this custom resource type to control updates on the **DepartmentEO** entity object.

Generate a Java class for **DepartmentEO** and override the `isAttributeUpdateable()` method. The ADF Business Component framework invokes `isAttributeUpdateable()` in an `oracle.jbo.server.EntityImpl` instance before updating each attribute. This example overrides the `isAttributeUpdateable()` method in `DepartmentEOImpl` to conditionally enable updates on attributes in the **DepartmentEO** entity instance. This example defaults the attribute values for newly created entity rows, and allows the user with special rights to override the defaults values. The algorithm used in this method is as follows:

1. It checks whether security is enabled for the attribute and if true continues with the next step.
2. Checks if the current row is new and if found true, continues with the next step. This method will take the default execution path if the current row is in the unmodified state.

3. Checks if the current user has the `ADFEntityRow` resource permission enabled with the **update** privilege. If all the previous conditions are met, then return true to enable updates on this attribute.

```
//In entity object implementation class (DepartmentEOImpl)

DataSecurityProviderManager _mDataSecurityMgr = null;
/**
 * Checks if the attribute is updateable.
 */
@Override
public boolean isAttributeUpdateable(int index) {
  DBTransactionImpl dbtransaction = (DBTransactionImpl)this.
getDBTransaction();
  DataSecurityProvider provider = _getDataSecurityProvider();
  if (provider == null) {
    return super.isAttributeUpdateable(index);
  }
  EntityCache ec = getEntityCache();
  AttributeDefImpl attrDef = (AttributeDefImpl)ec.
       getAttributeDef(index);
  String key = attrDef.getName();
  BindingPermissionDef permDef = attrDef.getPermissionDef();
  String privToCheck = (permDef == null ? null :
       permDef.findPrivilege(PermissionHelper.UPDATE_ACTION));
  //Variable privToCheck is null if no security has been
   // enabled on the entity attribute.
  //Note that Security can be enabled by choosing the Edit
  //Security option on the attribute context menu in
   //the Structure Window
  if (privToCheck == null) {
    return super.isAttributeUpdateable(index);
  }
  //check if attribute is new (insert case)
  if (getPostState() == STATUS_NEW ||
       getPostState() == STATUS_INITIALIZED) {
    //build ResourcePermission
    //type = ADFEntityRow,  Action = update
    String type = "ADFEntityRow";
    String entityName = this.getEntityDef().getName();
    String action = "update";
```

```
          SecurityContext securityCtx = ADFContext.getCurrent().
                getSecurityContext();
          ResourcePermission resourcePermission = new
                ResourcePermission(type, entityName, action);
          boolean userHasPermission =
                securityCtx.hasPermission(resourcePermission);

          if (userHasPermission) {
            return true;
          }
          return false;
        }
        return super.isAttributeUpdateable(index);

      }

      //The following is helper method to get DataSecurityProvider
      //Note that DBTransactionImpl:: getDataSecurityProvider() is //
      package private, hence not used
      DataSecurityProvider _getDataSecurityProvider() {
        if (_mDataSecurityMgr == null) {
          DBTransactionImpl dbtransaction =
                (DBTransactionImpl)this.getDBTransaction();
          _mDataSecurityMgr = new
                  DataSecurityProviderManager(dbtransaction);
        }
        return _mDataSecurityMgr.getDataSecurityProvider();
      }
```

Securing the data access layer

The data access layer in an application provides a simplified access to business data stored in persistent storage. ADF view objects build the data access layer for a Fusion web application. This section shows you how to enable authorization checks in the data access layer of a Fusion web application. Let us see the features provided by a view object to authorize each caller prior to returning the data requested by the client so that users are only able to see their own data.

Authorization check in view objects with secured entity object usages

The ADF framework has built-in support to secure data rows returned by a view object. When you execute a view object backed up by security-enabled entity object usages, it will invoke the `oracle.jbo.DataSecurityProvider` implementation configured in `adf-config.xml` during the query preparation phase in order to identify the `WHERE` clause fragment that needs to be added to the SQL statement for preventing unauthorized access to data rows. The default `DataSecurityProvider` implementation used by the ADF framework is `oracle.jbo.server.security.JAASDataSecurityProviderImpl`. If the authenticated user does not have read permission for the entity usages in a view object, `JAASDataSecurityProviderImpl` will append a dummy condition `"1 = 2"` to the `WHERE` clause of the query string preventing unauthorized access of data rows. The generated `SELECT` clause will look like the following:

```
SELECT <COLIMN_NAMES> FROM <TABLE_NAME> WHERE 1=2.
```

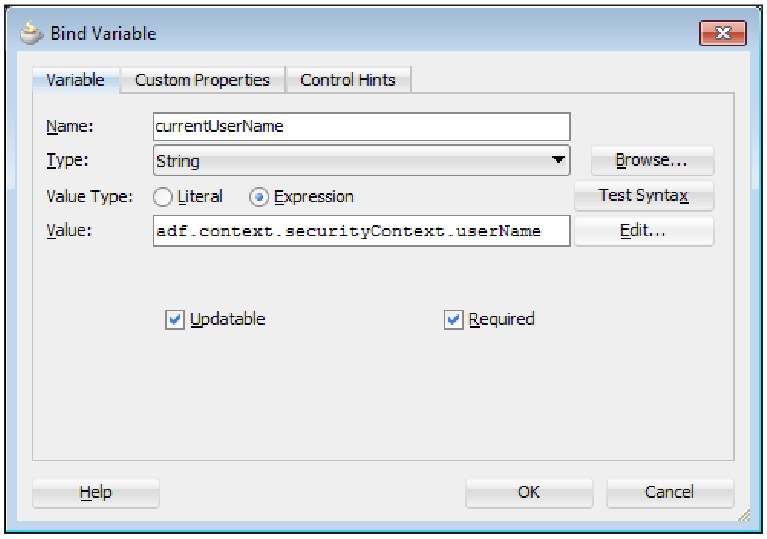
Referencing an authenticated username in the SQL WHERE clause

A common use case across many business applications involves preventing a logged in user viewing unauthorized business data. The target database table for such applications may typically have a column to hold the username and you will use this column in the SQL `WHERE` clause to pull up the records for the specified username.

To filter rows returned by a view object based on the logged in username, edit the query in the view object, and add the `WHERE` clause with a bind variable name as in the following example:

```
Where="EmployeeEO.NAME = :currentUserName".
```

Define the bind variable **currentUserName** as shown in the following screenshot. The value is set using the Groovy expression `adf.context.securityContext.userName` which points to the logged in user principal. Mark **Value Type** as **Expression** and select the **Required** option.



Name: currentUserName
Type: String
Value Type: Literal [✓ Expression]
Value: adf.context.securityContext.userName
✓ Updatable
✓ Required

Note that if you use this bind variable in the view criteria, you must *deselect* the **Required** checkbox.

Using a custom criteria adapter implementation to add an authorization check in a view object

The built-in support by the view object may be enough to secure the data read operations in most of the scenarios. However sometimes you may want to go beyond the declarative features and may want to add custom security clauses to the query at runtime. This section discusses a generic solution for such use cases. This example defines an empty view criteria on the view object and overrides the default query generation logic for this view criteria usage to build an appropriate SQL `WHERE` clause fragment. The following section explains this solution in detail.

1. The first step is to define an empty view criteria on the view object. Make sure you specify a unique name for the view criteria. The example used in this section names the dummy view criteria created for generating security clauses as `SecurityEnabledEmptyVC_`. We will use this name in the custom criteria adaptor class to identify the dummy view criteria added in the view object.

2. Once you define the view criteria, you may need to associate it with appropriate view object instances in the application module. To do this, perform the following steps.
3. Open the application module in the overview editor and select the **Data Model** tab.
4. Select the appropriate view object instance added to the application module in the **Data Model** list, then click on **Edit**.
5. In the **Edit View Instance** window, shuttle the dummy view criteria that you created to the **Selected** list.
6. The next step is to build a custom logic for generating the security predicate when you execute the view object instance with the special view criteria usage that we defined in step 1. This is explained as follows:

   The view object component allows you to override the default query generation implementation for the applied view criteria through a custom `oracle.jbo.CriteriaAdapter`. We will use this approach to inject custom security conditions in the query at runtime. If you need a quick brush up on this topic, refer back to the topic *Intercepting query generation for view criteria* in *Chapter 5*, *Advanced Concepts on Entity Objects and View Objects*.

   The following is an example that uses the custom `oracle.jbo.CriteriaAdapter` implementation for generating security predicates when you execute a view object instance with security enabled view criteria usage. This sample is kept simple to make the points clear. The `SecurityEnabledViewCriteriaAdapter` implementation used in this example returns the `WHERE` clause fragment with a custom security predicate as `<EO_Aliase>.USER_NAME = '<current_user_name>'` for view object instances with `SecurityEnabledEmptyVC_` usage. You can use a similar concept for more complex security checks in your application.

```
public class SecurityEnabledViewCriteriaAdapter extends
CriteriaAdapterImpl implements CriteriaAdapter {

    //Special view criteria name defined for injecting
    // security clause
    private static String SECVCNAME =
        "SecurityEnabledEmptyVC_";

    public SecurityEnabledViewCriteriaAdapter() {
        super();
    }

    /**
```

```
         * Generate a security predicate for the view
         * criteria(if conditions are met).
         * This example appends WHERE clause fragment
         * USER_NAME = '<surrent_username>' with the query
         * @param criteria a view criteria instance
         * @return a where clause fragment
         */
        public String getCriteriaClause(ViewCriteria criteria) {
            ViewObjectImpl vo = (ViewObjectImpl)criteria.
                getViewObject();
            if (isSecurityEnabled() && isSecureVC(criteria)) {
                String loggedInUser = ADFContext.getCurrent().
                    getSecurityContext().getUserName();
                ViewDefImpl voDef = (ViewDefImpl)vo.getDef();
                String securityClause =
                    voDef.getEntityUsages()[0].
                        getEntityDef().getAliasName() +
                        ".USER_NAME = '" + loggedInUser + "'";
                return securityClause;
            }
            return super.getCriteriaClause(criteria);
        }

        /**
         * Check if security is enabled for the application
         * @return
         */
        private boolean isSecurityEnabled() {
            SecurityContext secCtx = ADFContext.getCurrent().
                getSecurityContext();
            return secCtx.isAuthorizationEnabled();
        }

        /**
         * Check if the view criteris is meant for
         * generating security predicates
         * @param vc
         * @return
         */
        public static boolean isSecureVC(ViewCriteria vc) {
            return ((vc == null) ? false :
                (vc.getName() == null ? false :
                    vc.getName().equalsIgnoreCase(SECVCNAME)));
        }

    }
```

To hook this custom security enabled `CriteriaAdapter` implementation into a view object, override the `getCriteriaAdapter()` method in the desired view object implementation class as shown in the following code:

```
//In view object implementation class
/**
 * Return a custom CriteriaAdapter implementation
 * to generate where clause for ViewCriteria.
 *
 * @return Custom CriteriaAdapter implementation if
 * desired, or null.
 */
@Override
public CriteriaAdapter getCriteriaAdapter() {
  return new SecurityEnabledViewCriteriaAdapter();
}
```

7. To add the security enabled marker view criteria (`SecurityEnabledEmptyVC_`, defined in step 1) to a view object instance, open the appropriate application module in the overview editor. In the **Data Model** page, select the desired view object instance in the **Data Model** selected list and click on the **Edit** button. In the **Edit View Instance** dialog, shuttle the `SecurityEnabledEmptyVC_` to the selected list. Alternatively, you can call `applyViewCriteria(…)` on a view object to apply the view criteria at runtime.

   At runtime, when you execute a view object instance with view criteria usage, the framework invokes the `ViewObjectImpl::getCriteriaAdapter()` method to identify `CriteriaAdapter` used for generating the `WHERE` clause fragment for the view criteria. This example returns the `SecurityEnabledViewCriteriaAdapter` implementation class as a criteria adaptor. This class is responsible for generating a custom security predicate for view objects with the view criteria usage `SecurityEnabledEmptyVC_`.

Securing business service methods

Preventing unauthorized access to business services is very critical for any enterprise application. ADF security offers method permission definitions for the purpose of addressing such scenarios. Method permissions check if a user has the right to execute a method defined in the application. ADF security allows you to secure access to methods defined in the application through the `oracle.adf.share.security.authorization.MethodPermission` class.

To define method permissions in an application, perform the following steps:

1. Open the **jazn-data.xml** in the overview editor and select the **Resources Grant** tab.
2. Choose **ADF Method** as **Resource Type**. Add a new **Resource** value.
3. In the **Create Resource** dialog, specify a fully qualified class name along with a method name as value for the **Name** field. For example, if you are defining the method permission for `updateDeparment()` defined in the class `model.service.HRServiceAppModuleImpl`, the **Name** field is specified as `model.service.HRServiceAppModuleImpl.updateDeparment`.
4. Click on **OK** to save the changes and dispose of the dialog.

You can use security expressions to refer to the method permission definitions to control the display of action enabled UI components in a page. The ADF security framework also exposes APIs for checking the method permission which can be used to programmatically check the user privileges in the code. The following example illustrates the usage of method permissions in an application.

An example using method permissions

Let us see how method permissions can be used in an EL expression to control the display property of a command component.

The following is an example for a method permission definition in `jazn-data.xml`. This definition describes the `updateDeparment()` method in the `com.packtpub.adfguide.service.HRServiceAppModuleImpl` class:

```
<jazn-data ...>
...
 <resources>
  <resource>
  <name>
   model.service.HRServiceAppModuleImpl.updateDeparment
   </name>
  <display-name>updateDeparment</display-name>
  <description>updateDeparment</description>
  <type-name-ref>ADFMethodResourceType</type-name-ref>
  </resource>
 </resources>
</jazn-data>
```

When you grant method permissions to an application role, the IDE will generate a corresponding `<permission>` entry for the grantee in `jazn-data.xml` as follows:

```
<permission>
<class>
oracle.adf.share.security.authorization.MethodPermission
</class>
<name>
model.service.HRServiceAppModuleImpl.updateDeparment
</name>
<actions>invoke</actions>
</permission>
```

Using method permissions in an EL expression

The following component tag illustrates how the method permission that we defined in this example can be referenced through EL to enable or disable components based on the user rights for accessing the underlying operation:

```
<af:commandButton actionListener="#{bindings.updateDeparment.execute}"
  text="Update Department Details"
  disabled="#{!securityContext.userGrantedPermission['permissionCla
ss=oracle.adf.share.security.authorization.MethodPermission,target=
model.service.HRServiceAppModuleImpl.updateDeparment,action=invoke']}"
  id="cb6"/>
```

Using method permission APIs

The following code snippet illustrates the APIs for checking whether a user has access to a specific business method. The `oracle.adf.share.security.authorization.MethodPermission` instance used in this example refers to the permission settings for the `updateDeparment()` method that we defined at the beginning of this example.

```
//In application module implementation class

public void updateDeparment() {
    Permission permission = new MethodPermission
    ("model.service.HRServiceAppModuleImpl.updateDeparment",
    "invoke");
    SecurityContext securityCtx = ADFContext.getCurrent().
        getSecurityContext();
    boolean userHasPermission = securityCtx.
        hasPermission(permission);
    if(userHasPermission){
    //user is authorized to call this method
```

```
        //Add your business logic here
        _doUpdate();
       }
    }
```

Restricting data access using the virtual private database

While talking about data security, it is interesting to know the features offered by an Oracle database for meeting application security implementation. The **virtual private database** (**VPD**) is an Oracle database security feature, enabling row level or column level access control on database objects. At runtime, this VPD adds dynamic conditions while executing the user supplied queries to prevent unauthorized access of data.

If you use VPD for the purpose of securing your Fusion web application, most of the settings are done at database level. To pass user context information such as logged in username or enterprise name from the Java middle tier to the database session, you can override the `prepareSession(Session session)` method in the application module implementation class. This method will be invoked when an application module instance is associated with a user session. The `prepareSession()` method can have a custom stored procedure or other appropriate routines to pass user context data to a database session.

A detailed discussion on VPD is outside the scope of this book. To learn more about VPD refer to the topic *Virtual Private Database* which is available online at `http://www.oracle.com/technetwork/database/security/index-088277.html`.

Securing the user interface layer

Securing the UI layer simply means allowing users to see only what they have access to. Enabling security in a view layer of an application involves the following tasks:

- **Page authorization**: This task does not display the page if the logged in user does not have access to it.
- **Field authorization**: This disables or hides fields if the logged in user does not have access to them

- **Input validation**: This validates the data based on user privileges
- **User action authorization**: This disables or hides actionable components in the UI if the logged in user does not have access to the underlying business functionality

We have discussed how to authorize web pages under the topic *Providing resource grants* in this chapter. In this section, we will examine commonly used security expressions and APIs for the purpose of performing authorization checks within a page. You will use the security EL expressions to hide or disable UI components in a page based on the access rights of the user. While discussing security expressions, we will discuss corresponding security APIs as well. The security APIs are used to programmatically check the access rights in the business logic implementation.

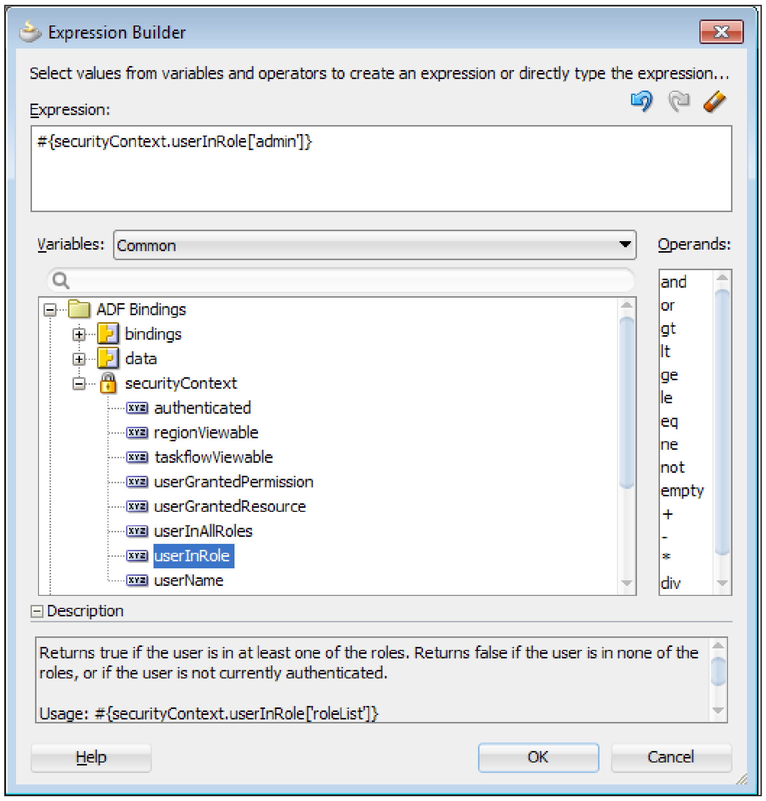
Using the expression builder to add security expressions to a user interface component

The JDeveloper IDE provides editor support for the purpose of adding security expressions to UI components in a JSF page.

To add a security expression to a user interface component, perform the following steps:

1. Open up the desired web page in the design editor and then select the appropriate component in the design view.
2. Go to the **Property Inspector** window and click on the down-arrow icon to the right-hand side of the property to which you want to add the expression (for example, the `disabled` property for `af:commandButton`). Choose **Expression Builder** from the context menu.
3. In the **Expression Builder** dialog, expand the **ADF Bindings | securityContext** node. Choose the appropriate expression displayed under the **securityContext** node. To know more about each expression, expand the **Description** node at the bottom of the expression editor.
4. Click on **OK** to select the expression.

The following screenshot displays the usage of the **Expression Builder** dialog for building security expressions to check whether the current user belongs to the admin role.



Returns true if the user is in at least one of the roles. Returns false if the user is in none of the roles, or if the user is not currently authenticated.

Usage: #{securityContext.userInRole['roleList']}

Commonly used ADF security expressions and security APIs in the UI layer

The commonly used security expressions and corresponding ADF security APIs are as follows:

- **Accessing logged in username**: The following EL expression can be used for accessing the authenticated username: `#{securityContext.userName}`.

  The following code snippet illustrates how to use the ADF security API for the purpose of accessing an authenticated username:

  ```
  SecurityContext securityCtx = ADFContext.getCurrent().
  getSecurityContext();
  String loggedInUserName=securityCtx.getUserName();
  ```

- **Is the user authenticated**: The following EL expression returns true if the user is authenticated: `#{securityContext.authenticated}`.

  The following is a code sample for the purpose of reading the authentication status for the current user:

  ```
  SecurityContext securityCtx = ADFContext.getCurrent().
  getSecurityContext();
  Boolean isUserAutheticated = securityCtx.isAuthenticated();
  ```

- **Is the user under a specific role**: The following EL expression returns true if the authenticated user is included in any of the roles specified as a comma separated list: `#{securityContext.userInRole['commaSeparatedRoleNames']}`.

  The following is a code sample to check if the user belongs to an `it-admin` role. Note that there is no API which takes comma speared role names:

  ```
  SecurityContext securityCtx = ADFContext.getCurrent().
  getSecurityContext();
  boolean isUserInRole =securityCtx.isUserInRole(
  "it-admin");
  ```

- **Is the user under all roles**: The following EL expression returns true if the authenticated user is in all of the roles specified as a comma separated list: `#{securityContext.userInAllRoles['commaSeparatedRoleNames']}`.

  The following method illustrates how to use the ADF security API to check if an authenticated user belongs to all roles passed as a comma separated argument:

  ```
  public boolean isUserInAllRoles(String commaSepartedRoles) {
    SecurityContext securityCtx = ADFContext.getCurrent().
         getSecurityContext();
    String[] assignedRoles = securityCtx.getUserRoles();

    String[] roles = commaSepartedRoles.split(",");
    List assignedRolesAsList = Arrays.asList(assignedRoles);
    for (String role : roles) {

      if (!assignedRolesAsList.contains(role)) {
        return false;
      }

    }
    return true;
  }
  ```

- **If the user has been granted permission for specific action**: The following EL expression returns true if the authenticated user has been granted the permission specified as argument: `#{securityContext.userGrantedPermission['permission']}`.

  The value for `permission` in this EL is a string containing a semicolon-separated concatenation of `permissionClass = qualifiedClassName;target = artifactName;action = actionName`. This EL essentially gathers the three pieces of information needed by **Java Platform Security** (**JPS**) to perform a **JAAS** permission check, returning a boolean value. Let us see a few examples using this expression to check access rights for a user.

  In the following example we are checking if the logged in user has view permission for `dept-task-flow-definition`: `#{securityContext.userGrantedPermission['permissionClass=oracle.adf.controller.security.TaskFlowPermission;target=/WEB-INF/dept-task-flow-definition.xml#dept-task-flow-definition; action=view']}`

  The following code snippet illustrates the APIs for checking the task flow permission:

```
TaskFlowId deptTaskFlowId = TaskFlowId.parse("/WEB-INF/dept-task-
flow-definition.xml#dept-ask-flow-definition");
ControllerContext controllerContext =
    ControllerContext.getInstance();
TaskFlowPermission taskFlowPermission=controllerContext.
getSecurity().
    getPermission(taskFlowId, TaskFlowPermission.VIEW_ACTION);
if(  ADFContext.getCurrent().
    getSecurityContext().hasPermission(taskFlowPermission)){
   //User has access to task flow
   //Add your business logic here
}
```

  The next example illustrates how you can use security EL to check whether the user has been granted permission for invoking the `updateDeparment()` method defined in the `HRServiceAppModuleImpl` class.

```
#{securityContext.userGrantedPermission['permissionClass=orac
le.adf.share.security.authorization.MethodPermission,target=
model.service.HRServiceAppModuleImpl.updateDeparment,action=in
voke']}
```

ADF provides simplified security expressions for checking the **view** permission on task flows and regions using `securityContext.taskflowViewable` and `securityContext.regionViewable` respectively. You really do not need to write lengthy expressions using `securityContext.userGrantedPermission` for checking **view** permissions on these resources. These simplified expressions are discussed as follows.

- **Task flow view permission**: The following is the EL expression for checking whether a user has view permission to the task flow: `#{securityContext.taskflowViewable['taskflow']}`

  The task flow in the previous EL expression is the WEB-INF node-qualified name of the task flow being accessed. This simplified version of the expression is `{securityContext.userGrantedPermission['permission']},` which presumes that the name of `permissionClass` to be used is `oracle.adf.controller.security.TaskFlowPermission` and the action to be used is `view`.

  The following example illustrates the usage of EL to check the task flow view permission for the logged in user:

  ```
  #{securityContext.taskflowViewable['/WEB-INF/dept-task-flow-
  definition.xml.xml#dept-task-flow-definition']}
  ```

  The following code snippet illustrates the APIs for checking the view permissions for a task flow for the logged in user:

  ```
  TaskFlowId deptTaskFlowId = TaskFlowId.parse("/WEB-INF/dept-task-
  flow-definition.xml#dept-ask-flow-definition");
  ControllerContext controllerContext = ControllerContext.
  getInstance();
  if(controllerContext.
      getSecurity().isViewAuthorized(taskFlowId)){
     //User has access to task flow
     //Add your business logic here
  }
  ```

- **Region view permission**: The following EL expression returns true if the authenticated user has view access to the page definition file passed as a parameter:

  ```
  #{securityContext.regionViewable['pagedef']}
  ```

The `pagedef` in this EL is the fully-qualified name of the page definition file associated with the web page being accessed. This simplified EL version presumes that the name of `permissionClass` to be used is `oracle.adf.share.security.authorization.RegionPermission` and the action to be used is view. The following is an example:

```
#{securityContext.regionViewable['com.packtpub.adfguide.ch14.
view.pageDefs.deptPageDef']}
```

The following example shows how to do this check programmatically:

```
RegionPermission perm = new RegionPermission(
    "com.packtpub.adfguide.ch14.view.pageDefs.deptPageDef",
         RegionPermission.VIEW_ACTION);
if( ADFContext.getCurrent().
    getSecurityContext().hasPermission(perm)){
   //User has access to region.
   //Add your business logic here
}
```

- **Resource permission**: The following EL expression returns true if the authorized user is granted the custom resource permission:

  ```
  #{securityContext.userGrantedResource['resource']}
  ```

  The `resource` in this EL expression is a semicolon-separated concatenation of `resourceName=<name>;resourceType=<type>;action=<action>`.

  The following is an example for the `securityContext.userGrantedResource` EL. This example returns true if the user is granted with the `update` action on the **ADFEntityRow** resource:

  ```
  #{securityContext.userGrantedResource['resourceName=DepartmentE
  O;resourceType= ADFEntityRow;action=update']}
  ```

  The following example will help you to understand the ADF security API for the purpose of performing a resource permission check in your code:

  ```
  String type = "ADFEntityRow";
  String entityName = DepartmentEOImpl.
                          getDefinitionObject().getName();
  String action = "update";
  SecurityContext securityCtx = ADFContext.getCurrent().
      getSecurityContext();
  ResourcePermission resourcePermission = new
  ResourcePermission(type, entityName, action);
  boolean userHasPermission = securityCtx.hasPermission(resourcePerm
  ission);
  if (userHasPermission) {
    //user has required permission granted
    //business logic goes here
  }
  ```

- **Accessing the enterprise name**: The EL expression for accessing the enterprise name for the logged in user is as follows: `#{data.adfContext.enterpriseName}`.

  The following is a code sample for reading the enterprise name:

  ```
  ADFContext adfContext= ADFContext.getCurrent();
  String enterpriseName = adfContext.getEnterpriseName();
  ```

Using ADF controller APIs to check user permissions

You can use ADF controller APIs to check access rights on resources such as task flows and views for a user.

- **View level authorization check**: The following EL expression returns true if the authenticated user has been granted access to the view specified in the expression:

  `#{controllerContext.security.activity['viewid'].viewAuthorized}`

  The following example returns true if the logged in user is granted view access to `DeptDetailsView`: `#{controllerContext.security.activity['DeptDetailsView'].viewAuthorized}`.

  You can use this expression where you need to skip the rendering of links that refer to a page which is not authorized to view by the current user.

- **Task flow authorization check**: The following EL expressions return true if the current user has been granted access to the task flow specified in the expression:

  i. `#{controllerContext.security.taskflow['taskflowid'].viewAuthorized}`

  ii. `#{controllerContext.security.outcome ['taskflowid'].viewAuthorized}`

  Functionality wise this is the same as the expression `#{securityContext.taskflowViewable['taskflow']}` that we discussed in the previous section.

  An example using the `controllerContext` expression is as follows:

  ```
  #{controllerContext.security.taskflow['/WEB-INF/dept-task-flow-
  definition.xml'].viewAuthorized}
  ```

EL expressions for checking user permissions on an entity object

We have discussed how to enable security for an entity object in the section *Securing data update operations*. You can use the following expressions in the JSF page to check user permissions for the underlying entity object.

- **Checking user permissions for entity objects**: The following binding EL returns true if the user has been granted privileges for performing specified operations on the underlying entity object:

  `#{row.hints.allows.<operation>}`

  The `<operation>` expression used in the previous EL could be the `read`, `update`, or `removeCurrentRow` operation that you set for the entity object.

  The following example uses EL to disable a button component in a table if the user is not allowed to update the underlying entity object:

  ```
  <af:table value="#{bindings.EmployeesInDepartment.
  collectionModel}" var="row" ...>
   <af:column headerText="Update Employee" id="c2">
    <af:commandButton text="Update" id="cb8" disabled="#{!row.hints.
  allows.update}"/>
   </af:column>
   ...
  </af:table>
  ```

> The expression `#{row.hints.allows.<operation>}` is evaluated only in the context of row collection. If you want to check the entity permission outside the context of row collection, you must use `#{securityContext.userGrantedResource ['resource']}` instead.

- **Checking user permissions for an entity object attribute**: The following binding EL expression returns true if the user has been granted privileges to perform specified operations on the underlying entity object attribute: `#{bindings.<attributeName>.hints.allows.<operation>}`.

  The following is an example:

  ```
  <af:inputText value="#{bindings.DepartmentId.inputValue}"
   ... ...
    disabled="#{bindings.DepartmentId.hints.allows.update}">
  </af:inputText>
  ```

To check the attribute permission in the context of row collection, you must use the following EL expression:

```
#{row.<attributeName>.hints.allows.<operation>}
```

Summary

In this chapter you learned how to use the JDeveloper IDE to declaratively secure a Fusion web application. You also learned various security features offered by the Oracle ADF security module to secure the view, bindings, and business services layers of an application.