This guide provides an introduction to Oracle Entitlements Server and describes how to create authorization policies, request authorization decisions and delegate administration using the available application programming interfaces (API). It also contains information regarding the Oracle Entitlements Server policy model, and how to use the API to create policy objects.
# Contents

**Preface** ......................................................................................................................................................... xi

- Audience ......................................................................................................................................................... xii
- Documentation Accessibility ........................................................................................................................... xii
- Related Documents ........................................................................................................................................... xii
- Conventions ..................................................................................................................................................... xii

## 1 Using the Policy Model

1.1 Examining Policy Elements ....................................................................................................................... 1-1
1.2 Composing A Simple Policy ....................................................................................................................... 1-2
1.3 Adding Fine Grained Objects to a Simple Policy ....................................................................................... 1-4
1.3.1 Creating an Application Role .................................................................................................................. 1-4
1.3.2 Defining A Role Mapping Policy ............................................................................................................ 1-5
1.3.3 Adding a Condition .............................................................................................................................. 1-6
1.3.4 Populating a Permission Set .................................................................................................................. 1-6
1.3.5 Building an Obligation .......................................................................................................................... 1-6
1.4 Using Roles to Implement Policy ............................................................................................................. 1-7

## 2 Constructing A Policy Programmatically

2.1 Using the Java API ........................................................................................................................................ 2-1
2.1.1 Creating a Policy Object ....................................................................................................................... 2-2
2.1.2 Modifying a Policy Object .................................................................................................................... 2-2
2.1.3 Deleting a Policy Object ...................................................................................................................... 2-2
2.1.4 Searching for Policy Objects ............................................................................................................... 2-2
2.2 Using the Data Types .................................................................................................................................. 2-3
2.3 Executing A Simple Policy ......................................................................................................................... 2-6
2.3.1 Accessing the Policy Store ................................................................................................................... 2-6
2.3.2 Creating an Application Policy ............................................................................................................. 2-7
2.3.3 Defining Resource Types ....................................................................................................................... 2-8
2.3.4 Instantiating a Resource ......................................................................................................................... 2-9
2.3.5 Associating Actions with the Resource ................................................................................................. 2-11
2.3.5.1 Using a ResourceEntry .................................................................................................................. 2-11
2.3.5.2 Using a ResourceNameExpression ............................................................................................... 2-12
2.3.6 Specifying a Policy Rule ....................................................................................................................... 2-13
2.3.7 Specifying the Principal ......................................................................................................................... 2-13
2.3.8 Defining the Policy .............................................................................................................................. 2-14
2.4 Creating Fine Grained Elements for a Simple Policy ......................................................... 2-16
  2.4.1 Creating Application Roles ......................................................................................... 2-16
  2.4.2 Creating Role Mapping Policies .................................................................................. 2-17
  2.4.3 Creating Attribute and Function Definitions ............................................................. 2-19
    2.4.3.1 Creating Attribute Definitions .............................................................................. 2-19
    2.4.3.2 Creating Custom Function Definitions ............................................................... 2-20
  2.4.4 Defining Permission Sets ......................................................................................... 2-21
  2.4.5 Defining a Condition ................................................................................................. 2-22
    2.4.5.1 Constructing a Boolean Expression ..................................................................... 2-25
    2.4.5.2 Constructing a Custom Function Expression .................................................... 2-26
  2.4.6 Adding Obligations .................................................................................................... 2-27
  2.5 Accessing Code Examples ............................................................................................ 2-28

3 Managing Policy Objects Programmatically
  3.1 Managing Policies Using Oracle Entitlements Server ...................................................... 3-1
  3.2 Using Scope Levels to Manage Policy Objects ............................................................... 3-1
    3.2.1 Managing Objects Created at the PolicyStore Scope .............................................. 3-3
    3.2.2 Managing Objects Within the ApplicationPolicy Scope .......................................... 3-3
    3.2.2.1 Managing PolicyDomainEntry Objects .............................................................. 3-4
    3.2.2.2 Managing ResourceTypeEntry Objects ............................................................... 3-5
    3.2.2.3 Managing and Granting AppRoleEntry Objects ................................................. 3-5
    3.2.2.4 Managing Role Mapping Policy (RolePolicyEntry) Objects ............................... 3-6
    3.2.2.5 Managing AttributeEntry and FunctionEntry Objects ...................................... 3-7
      3.2.2.5.1 Managing AttributeEntry Objects ................................................................. 3-7
      3.2.2.5.2 Managing FunctionEntry Objects ................................................................ 3-8
    3.2.2.6 Managing ResourceEntry Objects ....................................................................... 3-9
    3.2.2.7 Managing Permission Sets ................................................................................... 3-10
    3.2.2.8 Managing the Policy ............................................................................................ 3-11
    3.2.3 Managing Objects within the PolicyDomainEntry Scope ....................................... 3-12

4 Distributing Policies
  4.1 Understanding Policy Distribution ............................................................................... 4-1
    4.1.1 Using a Centralized Policy Distribution Component ............................................. 4-1
    4.1.2 Using a Local Policy Distribution Component ...................................................... 4-2
  4.2 Defining Distribution Modes ....................................................................................... 4-3
    4.2.1 Controlled Distribution .......................................................................................... 4-3
    4.2.2 Non-Controlled Distribution .................................................................................. 4-4
  4.3 Creating Security Module Configurations and Bindings ............................................ 4-4
    4.3.1 Managing Security Module Configurations ......................................................... 4-5
    4.3.2 Managing Security Module Bindings .................................................................... 4-6
  4.4 Initiating Policy Distribution ....................................................................................... 4-7

5 Delegating Policy Administration
  5.1 Delegating Administration............................................................................................. 5-1
  5.2 Managing Scope and Delegating Granularity .............................................................. 5-2
  5.3 Assigning Permissions ................................................................................................. 5-3
7.2.1.1 Step 1 Write the Custom Code ................................................................. 7-6
7.2.1.2 Step 2. Compile the Source Code and Include the Class in a JAR File .... 7-8
7.2.1.3 Step 3. Add the Jar File to CLASSPATH of WebLogic Domain on which the Application is Deployed 7-8
7.2.1.4 Step 4. Restart the Servers ................................................................. 7-8
7.2.2 Using InspectableFunction For Metadata Information ........................... 7-9

8 Using the JSP Standard Tag Library

8.1 Using the Tag Library ............................................................................. 8-1
8.2 Defining the Functional Tags ................................................................. 8-1
8.2.1 isAccessAllowed Tag ......................................................................... 8-1
8.2.2 isAccessNotAllowed Tag ................................................................. 8-3
8.2.3 getUserRoles Tag ............................................................................... 8-5
8.2.4 isUserInRole Tag ............................................................................... 8-6
8.3 Defining the Assistant Tags ................................................................. 8-7
8.3.1 setSecurityContext Tag ................................................................. 8-8
8.3.2 attribute Tag ..................................................................................... 8-8
8.3.3 then/else Tags ................................................................................... 8-9

Index
List of Examples

2–1 Using createApplicationPolicy() Method ................................................................. 2-8
2–2 Using the createResourceType() Method ................................................................. 2-8
2–3 Using createResource() Method .............................................................................. 2-10
2–4 Building a ResourceActionsEntry with ResourceEntry ........................................ 2-11
2–5 Building a ResourceActionsEntry with ResourceNameExpression ....................... 2-12
2–6 Create a PolicyRuleEntry ......................................................................................... 2-13
2–7 Using createPolicy() Example ............................................................................... 2-15
2–8 Creating an Application Role .................................................................................. 2-16
2–9 Assigning Principals to an Application Role ......................................................... 2-17
2–10 Applying Application Role Hierarchies ............................................................... 2-17
2–11 Using the createRolePolicy() Method ................................................................. 2-18
2–12 Creating a Dynamic Attribute Definition ............................................................ 2-20
2–13 Creating a Custom Function Definition ............................................................... 2-21
2–14 Building a PermissionSetEntry ............................................................................ 2-22
2–15 Defining a BooleanExpressionEntry ..................................................................... 2-23
2–16 Building a BooleanExpressionEntry ..................................................................... 2-24
3–1 Definition of a Policy Store in jps-config.xml ...................................................... 3-1
3–2 Using deleteApplicationPolicy() Method ............................................................. 3-3
3–3 Using deletePolicyDomain() Method .................................................................... 3-4
3–4 Using modifyPolicyDomain() Method .................................................................... 3-4
3–5 Using getPolicyDomain() Method .......................................................................... 3-4
3–6 Using the deleteResourceType() Method .............................................................. 3-5
3–7 Using deleteAppRole() Method .............................................................................. 3-6
3–8 Using the deleteRolePolicy() Method ................................................................... 3-7
3–9 Using the modifyRolePolicy() Method .................................................................. 3-7
3–10 Using the getAttribute() Method ....................................................................... 3-7
3–11 Using the deleteAttribute() Method .................................................................... 3-8
3–12 Using the getFunction() Method ......................................................................... 3-8
3–13 Using the deleteFunction() Method .................................................................... 3-8
3–14 Using the getResource() Method ......................................................................... 3-9
3–15 Using deleteResource() Method .......................................................................... 3-9
3–16 Using modifyResource() Method ......................................................................... 3-10
3–17 Modifying a PermissionSetEntry .......................................................................... 3-10
3–18 Using the deletePermissionSet() Method ............................................................ 3-10
3–19 Using modifyPolicy() Method ............................................................................ 3-11
3–20 Using deletePolicy() Method ............................................................................... 3-11
4–1 Using the createSecurityModule() Method .......................................................... 4-4
4–2 Using the bindSecurityModule() Method .............................................................. 4-5
4–3 Using the getSecurityModule() Method ................................................................ 4-5
4–4 Using the deleteSecurityModule() Method ........................................................... 4-6
4–5 Using the getBoundSecurityModules() Method ..................................................... 4-6
4–6 Using the getBoundApplications() Method .......................................................... 4-6
4–7 Using the unbindSM() Method ................................................................................ 4-7
4–8 Using the distributePolicy() Method .................................................................... 4-7
5–1 Using deleteAdminRole() Method .......................................................................... 5-6
5–2 Using getAdminRole() Method ............................................................................... 5-6
5–3 Using createPolicyDomain() Method .................................................................... 5-7
6–1 Using Authenticated Subject in Java PEP API Request ......................................... 6-7
6–2 Using WebLogic Server Subject with Java PEP API Request ................................. 6-8
6–3 Using Websphere Application Server Subject with Java PEP API Request ............... 6-8
6–4 Using OPSS Subject with Java PEP API Request .................................................. 6-9
6–5 Requesting Java PEP API Authorization Against a Resource .................................. 6-9
6–6 Making PEP API Request and Parsing Response .................................................. 6-10
6–7 Returned Obligations from Example 6–6 ............................................................... 6-10
List of Figures

1–1 Policy Components Mapped to Policy Objects................................................................. 1-2
4–1 Using Oracle Entitlements Server Policy Distribution Component.......................... 4-2
4–2 Using the Security Module Policy Distribution Component ...................................... 4-2
5–1 The Administration Role Model....................................................................................... 5-2
6–1 Relationship Between Open AZ API and PEP API...................................................... 6-2
List of Tables

2–1  Using the Complex SearchQuery Parameters ............................................................... 2-3
2–2  Data Types Descriptions ............................................................................................ 2-4
2–3  Functional Categories for Data Types ....................................................................... 2-5
2–4  Matching ResourceNameExpression Objects ....................................................... 2-12
2–5  Examples of ResourceNameExpression .................................................................. 2-12
2–6  Specifying a Principal Programmatically .............................................................. 2-14
5–1  Resource Name Options ............................................................................................ 5-5
6–1  Understanding the PEP API Authorization Request Types .................................... 6-3
6–2  PEP API Query Types .............................................................................................. 6-6
7–1  Methods in AttributeRetrieverV2 Interface ........................................................... 7-3
8–1  isAccessAllowed Tag Definition ............................................................................... 8-2
8–2  isAccessNotAllowed Tag Definition ....................................................................... 8-4
8–3  getUserRoles Tag Definition .................................................................................... 8-5
8–4  isUserInRole Tag Definition .................................................................................... 8-6
8–5  setSecurityContext Tag Definition ......................................................................... 8-8
8–6  attribute Tag Definition .......................................................................................... 8-8
Preface

This guide provides an introduction to Oracle Entitlements Server and describes how to create authorization policies, request authorization decisions and delegate administration using the available application programming interfaces (API). It also contains information regarding the Oracle Entitlements Server policy model, and how to use the API to create policy objects.

Audience

This document is intended for engineers who use Oracle Entitlements Server development tools to control access to an organization’s protected resources. This might involve programmatically requesting an authorization decision, creating an authorization or role mapping policy, developing custom Security Modules and Attribute Retrievers, and managing policy objects.

Documentation Accessibility

For information about Oracle’s commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following guides in the Oracle Entitlements Server documentation set:

- Oracle Fusion Middleware Release Notes
- Oracle Fusion Middleware Installation Guide for Oracle Identity Management
- Oracle Fusion Middleware Administrator’s Guide for Oracle Entitlements Server
- Oracle Fusion Middleware Management Java API Reference for Oracle Entitlements Server
- Oracle Fusion Middleware PDP Extension Java API Reference for Oracle Entitlements Server
## Conventions

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td>monospace</td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
Oracle Entitlements Server uses a model to define the elements that comprise a policy and how to use those elements to create a policy. The Oracle Fusion Middleware Administrator’s Guide for Oracle Entitlements Server has detailed information on the policy model. It includes a glossary of the model’s components and a use case for implementing policy. This chapter contains information on how the Oracle Entitlements Server policy model is implemented using the Management API. It contains the following sections.

- Section 1.1, "Examining Policy Elements"
- Section 1.2, "Composing A Simple Policy"
- Section 1.3, "Adding Fine Grained Objects to a Simple Policy"
- Section 1.4, "Using Roles to Implement Policy"

### 1.1 Examining Policy Elements

A policy is created to bestow an effect (GRANT or DENY) upon a request for a protected target resource based on the profile of the requesting principal. From a high level, the policy defines an association between an effect, a principal, the target resource, the resource’s allowed actions and an optional condition. A policy is applicable to a request for access if the parameters in the request match those specified in the policy. Consider the syntax of this policy:

```
GRANT the SupportManagerEast role MODIFY access to the Incidents servlet
    if the request is made from an IP address of 229.188.21.21; return Obligation
    ('send log message if access is granted')
```

Figure 1–1 illustrates how these elements map to policy-related objects (in the policy model) that can be used to create policies programmatically.
An effect (PolicyRuleEntry.EffectType) and an optional condition (RuleExpressionEntry) are defined in a policy rule (PolicyRuleEntry). The target resource (ResourceEntry) and the actions that can be performed on it are defined in a ResourceActionsEntry. The requesting user, group or role is defined as the principal (PrincipalEntry) and the principal has been assigned a role defined in an AppRoleEntry. The optional obligation (ObligationEntry) specifies information returned to the caller with the decision. It will be evaluated during enforcement of the decision (rather than during evaluation of the decision); the information may or may not be used by the caller, or affect the decision itself. These programmatic objects are stored in an instance of a policy store (PolicyStore). For more information, see Section 1.2, "Composing A Simple Policy" and Section 2.3.1, "Accessing the Policy Store." Additionally:

- Authorization Policies define rules that control access to both application software components and application business objects. See Chapter 2, "Constructing A Policy Programmatically" for more information.
- Role Mapping Policies define rules that control how principals are granted or denied roles. See Section 1.3.2, "Defining A Role Mapping Policy" for information.

Chapter 2, "Constructing A Policy Programmatically" has more information.

### 1.2 Composing A Simple Policy

Composing a simple Authorization Policy requires that the elements (or policy objects) be created in a particular order. For example, a ResourceEntry object can only be created after defining a ResourceTypeEntry object. A simple policy can be composed by following the sequence described below.

1. Access the policy store.

A PolicyStore object represents the entire policy store. All policy management activity can be initiated only by an authenticated user with the administrative rights to retrieve a handle to the policy store and manage the policies. The user must be assigned to at least one Administrative Role. Errors will be returned for any methods the role is not authorized to call. For more information, see Section 2.3.1, "Accessing the Policy Store."
2. Create an ApplicationPolicy.

An ApplicationPolicy object is a child of the PolicyStore object and should be created as the overall container for policies and related information that secure the components of a particular application. You may create as many ApplicationPolicy objects as needed although it is recommended that only one is created for each application to be secured. After using the createApplicationPolicy method, the ApplicationPolicy object handle is returned. For more information, see Section 2.3.2, "Creating an Application Policy."

3. Create a ResourceTypeEntry.

A ResourceTypeEntry object specifies one or more resource attributes, and definitions of all possible valid actions that can be performed on a particular kind of resource. The actions can be standard actions (GET and POST to a URL) or custom actions on a business object (transfer to or from a bank account). Consider the following ResourceTypeEntry objects and their valid actions:

- A text file may support Read, Write, Copy, Edit, and Delete.
- A checking account application may support deposit, withdrawal, view account balance, view account history, transfer to savings, and transfer from savings.

Actions will be granted or denied when accessing a protected ResourceEntry instance created from the ResourceTypeEntry. To create a ResourceTypeEntry, call the ResourceTypeManager which provides methods to create, read, update and modify the object. For more information, see Section 2.3.3, "Defining Resource Types."

4. Instantiate a ResourceEntry from the ResourceTypeEntry.

A specific protected target (ResourceEntry) will be instantiated from a ResourceTypeEntry object. The ResourceManager provides methods to create, read, update and delete a ResourceEntry. A ResourceEntry object represents a secured target (for example, an application), references a ResourceTypeEntry, and is created under a PolicyDomainEntry object. If no PolicyDomainEntry object is specified, it is created under the default PolicyDomainEntry object. For more information, see Section 2.3.4, "Instantiating a Resource."

Note: The PolicyDomainEntry is an organizational concept. An instance of the default PolicyDomainEntry cannot be retrieved. To retrieve objects within the default PolicyDomainEntry, get the ApplicationPolicy and use the objects' Manager interfaces.

5. Associate the applicable actions with the instantiated ResourceEntry using the ResourceActionsEntry interface.

Build a ResourceActionsEntry object to define the actions that can be performed on a ResourceEntry object. The set of actions defined in a ResourceActionsEntry
6. Build the PolicyEntry.

This includes:

a. Specifying the effects (GRANT or DENY) in a PolicyRuleEntry object.

See Section 2.3.6, "Specifying a Policy Rule" for more information.

b. Specifying a User or Group as the policy principal in a PrincipalEntry object.

See Section 2.3.7, "Specifying the Principal" for more information. You can also specify an Application Role as the policy principal. See Section 1.3.1, "Creating an Application Role" for more information.

c. Using the ResourceActionsEntry object containing the target resource instance and applicable actions.

See Section 2.3.5, "Associating Actions with the Resource" for more information.

d. Calling the PolicyManager and creating the PolicyEntry.

See Section 2.3.8, "Defining the Policy" for more information.

This sequence, and information on the creation of the policy objects, is reiterated in Chapter 2, "Constructing A Policy Programmatically" with additional information. Programming details regarding the management (including retrieval, modification and deletion) of policy objects is in Chapter 3, "Managing Policy Objects Programmatically."

1.3 Adding Fine Grained Objects to a Simple Policy

Section 1.2, "Composing A Simple Policy" documented the minimum components needed to create a policy. The following sections contain information on the objects that can be added to a simple policy to make it more fine grained.

- Section 1.3.1, "Creating an Application Role"
- Section 1.3.2, "Defining A Role Mapping Policy"
- Section 1.3.3, "Adding a Condition"
- Section 1.3.4, "Populating a Permission Set"
- Section 1.3.5, "Building an Obligation"

Additional programmatic information regarding the creation of these objects is in Chapter 2, "Constructing A Policy Programmatically." Additional programmatic information regarding the retrieval, modification, and deletion of these objects is in Chapter 3, "Managing Policy Objects Programmatically."

1.3.1 Creating an Application Role

An Application Role is a collection of users, groups, and other Application Roles. For example, you might grant an Application Role all privileges necessary for a given target application. After the Application Role is created, it can be assigned statically to
a user by granting the user membership in the role. It can also be assigned
dynamically by referencing the roles in a Role Mapping Policy which will, in turn,
grant the policy's principals the permissions defined in the policy itself. An
Application Role can be assigned to an enterprise user, group, or role in an identity
store, or another Application Role in the policy store. One target application may have
several different roles, with each role assigned a different set of privileges for more
fine-grained access.

Application Roles are defined at the ApplicationPolicy level (thus, its name). The
AppRoleEntry object represents the Application Role. The AppRoleManager provides
the methods to create, delete, modify and search for application roles as well as
methods to grant and revoke membership in the role. Membership can be granted
statically through the use of the grantAppRole() method or dynamically with a Role
Mapping Policy. A Role Mapping Policy assigns the role to users and an Authorization
Policy defines the role's access rights.

Note: For more information on Role Mapping Policies, see
Section 1.3.2, "Defining A Role Mapping Policy."

Application Roles use role inheritance and hierarchy. The inheritance pattern is such
that an enterprise user or group or an identity role assigned to an Application Role
(using a Role Mapping Policy) also inherits any child roles as long as it is not
prohibited by other Role Mapping Policies. For example, if Application Role 1 is
granted to Application Role 2, Application Role 2 is a member of Application Role 1.
Thus, all subjects defined as Application Role 2 are also defined as Application Role 1
(if not prohibited by other Role Mapping Policies) through inheritance. When an
AppRoleEntry is referenced as a policy principal, access to the resource for all users
assigned the role is governed by the policy.

Note: An Application Role cannot be assigned to another
Application Role with a Role Mapping Policy. Membership between
Application Roles can only be defined statically.

For more information, see Section 2.4.1, "Creating Application Roles."

1.3.2 Defining A Role Mapping Policy

Access to a protected resource can be granted by defining the resource and the specific
users or groups that can access it in an Authorization Policy. But access can also be
granted by defining an Application Role, setting the protected resource and
Application Role in an Authorization Policy and creating a Role Mapping Policy to
dynamically determine the users, prior to authorization, at runtime.

As documented in Section 1.3.1, "Creating an Application Role," membership to an
Application Role can be granted statically through the use of the grantAppRole() method or dynamically with a Role Mapping Policy (RolePolicyEntry). An Application
Role, referenced as a Principal in a Role Mapping Policy, could grant a user access to
the defined resources but the Role Mapping Policy needs to be resolved before an
authorization decision is reached. The resolution answers the question Can the user
requesting access be assigned this Application Role? Once a request for access to a resource
is received, the Authorization Policies that apply are retrieved and evaluated. If the
policy references any Application Roles as the principal, they must be evaluated before
the access decision is made.
You can also apply restrictions that limit access to the resource by defining conditions on the Role Mapping Policy and/or the Authorization Policy - such as the time of day or the day of the week. See Section 1.3.3, "Adding a Condition" for more information. Section 1.4, "Using Roles to Implement Policy" and Section 2.4.2, "Creating Role Mapping Policies" contains more information on Role Mapping Policies.

1.3.3 Adding a Condition

A Condition can be added to either an Authorization Policy or a Role Mapping Policy as a way of setting an additional constraint on the policy. A Condition is written in the form of an expression that resolves to true or false (boolean) and has one of the following outcomes:

- If the expression resolves to true, the policy condition is satisfied and the PolicyRuleEntry is applicable.
- If the expression does not resolve to true, the policy is not applicable.

Conditions can be complex combinations of boolean expressions that test the value of some user, resource, or system attribute or they can be custom Java evaluation functions that evaluate complex business logic. To create a Condition in which you want to define an attribute and/or function, call the ExtensionManager and use the available methods to create one of the following objects for reference in the policy:

- AttributeEntry (a name/value pair that can be dynamically added to a policy rule)
- FunctionEntry (externally implemented logic)

Either can be added to a PolicyRuleEntry as a means of setting a Condition during policy evaluation. For more information, see Section 2.4.5, "Defining a Condition."

1.3.4 Populating a Permission Set

A ResourceActionsEntry object associates a specific protected target (Resource) with the action(s) that can be performed on it. The ResourceActionsEntry object is specified when creating a simple Authorization Policy or you can build a more complex policy by populating a PermissionSetEntry object (also referred to as the entitlement) with one or more ResourceActionEntry objects.

To populate, call the PermissionSetManager, instantiate a PermissionSetEntry and add one or more ResourceActionsEntry objects. The PermissionSetEntry is then referenced in a PolicyEntry object. For more information, see Section 2.4.4, "Defining Permission Sets."

1.3.5 Building an Obligation

An Obligation specifies optional information that is returned to the calling application with the access decision. This information may or may not be taken into account during policy enforcement based on settings defined by the application. The Obligation information is returned with the allowed policy effect (GRANT or DENY). For example, the reason a request for access has been denied might be returned as an Obligation. A different type of Obligation might involve sending a message; for example, if a certain amount of money is withdrawn from a checking account, send a text message to the account holder’s registered mobile phone.

**Note:** If a Condition evaluates to false, Obligations are not sent to the caller.
To specify an Obligation, build an ObligationEntry object. This object contains a set of attributes that form the arguments of the Obligation. The ObligationEntry is then referenced in a PolicyEntry object. For more information, see Section 2.4.6, "Adding Obligations."

1.4 Using Roles to Implement Policy

As documented in Section 1.3.2, "Defining a Role Mapping Policy," when users and groups are mapped to Application Roles, the mapping can be static (using direct role membership) or dynamic (using a Role Mapping Policy). A Role Mapping Policy contains the Principal (User, Group), an optional Target (resource, resource name expression) and (optionally) a Condition. Roles can also be mapped to access rights using an Authorization Policy. An Authorization Policy can contain the Principal (User, Group, Application Role), a Target (resource, entitlement set, resource name expression), actions that can be performed on the target, and (optionally) a Condition and Obligation. The following happens during authorization evaluation:

1. Based on the Principal, a list of Application Roles is determined by checking static role membership and applicable Role Mapping Policies.

2. Based on the Principal and resulting list of Application Roles, a list of Authorization Policies is evaluated to find any that are applicable. An applicable policy is based on the principal, target matching and condition evaluation.

3. Evaluation results are based on the DENY overrides combining algorithm.

For more information, see Section 2.4.1, "Creating Application Roles" and Section 2.4.2, "Creating Role Mapping Policies."
Constructing A Policy Programmatically

Oracle Entitlements Server contains Java application programming interfaces (API) for creating policies and policy objects programmatically. This chapter contains information on how to create these various policy objects using the API. It contains the following sections.

- Using the Java API
- Using the Data Types
- Executing A Simple Policy
- Creating Fine Grained Elements for a Simple Policy
- Accessing Code Examples

Note: For information on the corresponding WebLogic Scripting Tool (WLST) commands, see Oracle Fusion Middleware Administrator’s Guide for Oracle Entitlements Server.

2.1 Using the Java API

The Oracle Entitlements Server Java API can be used to construct, manage (read, modify, delete) and search for the policy objects discussed in Chapter 1, "Using the Policy Model." Policy definitions are constructed from these policy objects. A policy object is generally any interface that ends in Entry. The oracle.security.jps.service.policystore.info package comprises most of the policy objects including (but not limited to) the PolicyEntry, AppRoleEntry, and PermissionSetEntry. The oracle.security.jps.service.policystore.info.resource package comprises the ResourceEntry, ResourceTypeEntry and the ResourceActionsEntry. To construct or manage a policy object, you must:

1. Get a handle to the policy store.
   See Section 2.3.1, "Accessing the Policy Store."

2. Retrieve the application policy under which the object will be (or has been) created.
   See Section 2.3.2, "Creating an Application Policy."

Note: This may or may not entail the retrieval of a policy domain. See Chapter 5, "Delegating Policy Administration" for more information.
3. Retrieve an instance of the appropriate entity manager interface. A policy object is constructed and managed using an entity manager.

   The `oracle.security.jps.service.policystore.entitymanager` package comprises all interfaces including (but not limited to) the `ResourceManager`, `PolicyManager`, `AppRoleManager`, and `PermissionSetManager`.

The following sections document more specifically how the API can be used to perform specific operations on policy objects.

- Section 2.1.1, "Creating a Policy Object"
- Section 2.1.2, "Modifying a Policy Object"
- Section 2.1.3, "Deleting a Policy Object"
- Section 2.1.4, "Searching for Policy Objects"

For more detailed information, see one or both of the Oracle Entitlements Server Java API Reference guides.

- Oracle Fusion Middleware Management Java API Reference for Oracle Entitlements Server
- Oracle Fusion Middleware PDP Extension Java API Reference for Oracle Entitlements Server

### 2.1.1 Creating a Policy Object

To create a particular policy object, get a handle to the policy store and an instance of the applicable entity manager interface and use the `create` method. Policy objects have common elements that should be defined when they are being created including a Name, Display Name and Description. Additional elements that are specific to the type of object being created must also be defined. See Section 2.3, "Executing A Simple Policy" and Section 2.4, "Creating Fine Grained Elements for a Simple Policy" for examples of the `create` method and descriptions of its parameters.

### 2.1.2 Modifying a Policy Object

To modify a particular policy object, get a handle to the policy store and retrieve the object, either by creating a new one or searching for an existing one. An instance of the object will be placed in memory. Use the object's methods to modify the in-memory instance; you can call one or more as necessary. After completing the modifications, get an instance of the object's `Manager` interface and use the `modify` method, passing to it a reference to the in-memory object. This will propagate the changes to the object itself in the policy store. See Chapter 3, "Managing Policy Objects Programmatically" for examples of these operations and descriptions of their parameters.

### 2.1.3 Deleting a Policy Object

To delete a policy object, get a handle to the policy store and an instance of the applicable entity manager interface. Pass the object's defined Name to the manager's `delete` method to remove it. Additionally, some objects allow cascade removal. See Chapter 3, "Managing Policy Objects Programmatically" for more information.

### 2.1.4 Searching for Policy Objects

Searches are often required to retrieve policy objects referenced in policy definitions. To search for policy objects, use a simple query or a complex query. Each `Manager` interface has a singular and plural `get` method for each type of query, respectively. Use
the singular `get` method to search for, and retrieve, a specific policy object by passing the object’s defined Name. Use the plural `get` method to retrieve multiple objects using a complex query. With the plural `get` method, pass search criteria to it using the appropriate `SearchQuery` class as defined in the `oracle.security.jps.service.policystore.search` package. Table 2–1 documents the parameters and descriptions of the generic `SearchQuery` classes.

**Table 2–1 Using the Complex SearchQuery Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>policy_object.SEARCHPROPERTY</code></td>
<td>An enum in which the properties used to perform the query are defined. May include Name, Display Name, Description and others that vary by object type. For the permitted properties of a particular search type, see the Java API Reference.</td>
</tr>
<tr>
<td>negation</td>
<td>A boolean that takes as a value either <code>true</code> or <code>false</code>. If true, the NOT operator is applied to the search.</td>
</tr>
<tr>
<td>operator</td>
<td>An enum that defines the <code>ComparatorType</code> as one of the following:</td>
</tr>
<tr>
<td></td>
<td>■ EQUALITY</td>
</tr>
<tr>
<td></td>
<td>■ GREATER THAN</td>
</tr>
<tr>
<td></td>
<td>■ GREATER THAN OR EQUAL TO</td>
</tr>
<tr>
<td></td>
<td>■ LESS THAN</td>
</tr>
<tr>
<td></td>
<td>■ LESS THAN OR EQUAL TO</td>
</tr>
<tr>
<td>search string</td>
<td>Takes as a value the string used for the search.</td>
</tr>
<tr>
<td></td>
<td>■ If the value is null, the match must be ANY.</td>
</tr>
<tr>
<td></td>
<td>■ If populated, the algorithm matches the value against those being searched.</td>
</tr>
<tr>
<td><code>SearchQuery.MATCHER</code></td>
<td>An enum that defines how the search string is matched against the values being searched. It should define one of the following:</td>
</tr>
<tr>
<td></td>
<td>■ ANY — Any non-NULL value satisfies the search string. (If the search string is NULL, the match must be ANY.) Use this to retrieve all instances of the object type.</td>
</tr>
<tr>
<td></td>
<td>■ BEGINS_WITH — The object property must begin with the search string.</td>
</tr>
<tr>
<td></td>
<td>■ CONTAINED_IN — The object property must contain the search string.</td>
</tr>
<tr>
<td></td>
<td>■ ENDS_WITH — The object property must end with the search string.</td>
</tr>
<tr>
<td></td>
<td>■ EXACT — The object property must be exactly the same as the search string.</td>
</tr>
</tbody>
</table>

See Chapter 3, "Managing Policy Objects Programmatically" for more information on these search operations. See the Oracle Entitlements Server Java API Reference for `SearchQuery` parameter information specific to the particular policy object.

### 2.2 Using the Data Types

Policy store support for additional data types has been added to Oracle Entitlements Server to make it compliant with the XACML 2.0 standard. The data type used is dependent on the type of data itself and how Oracle Entitlements Server will interpret it. For example, if you have an IP address and want Oracle Entitlements Server to use IP address semantics in handling it for masking, use the `OpssIPAddress` data type. If `OpssString` is used, Oracle Entitlements Server wouldn’t know that the string needs to
be treated as an IP address. Table 2–2 describes the data types.

### Table 2–2 Data Types Descriptions

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY_URI</td>
<td>Represents a Uniform Resource Identifier (URI) with an optional fragment identifier.</td>
</tr>
<tr>
<td>BASE64_BINARY</td>
<td>Represents arbitrary Base64-encoded binary data.</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Represents true or false.</td>
</tr>
<tr>
<td>DATE</td>
<td>Values may be viewed as objects with integer-valued year, month, day, hour and minute properties, a decimal-valued second property, and a boolean time-zoned property.</td>
</tr>
<tr>
<td>DATE_TIME</td>
<td>Values may be viewed as objects with integer-valued year, month, day, hour and minute properties, a decimal-valued second property, and a boolean time-zoned property.</td>
</tr>
<tr>
<td>DAYTIME_DURATION</td>
<td>Represents a duration of time as an object where the coordinates designate the Gregorian year, month, day, hour, minute, and second components.</td>
</tr>
<tr>
<td>DNS_NAME</td>
<td>Represents a Domain Name Service (DNS) host name, with optional port or port range.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Represents the IEEE double-precision 64-bit floating point type.</td>
</tr>
<tr>
<td>HEX_BINARY</td>
<td>Represents arbitrary hex-encoded binary data.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Represents a finite-length sequence of decimal digits with an optional leading sign. If the sign is omitted, positive numerals are assumed.</td>
</tr>
<tr>
<td>IP_ADDRESS</td>
<td>Represents an Internet Protocol (IP) address.</td>
</tr>
<tr>
<td>RFC822_NAME</td>
<td>Represents an e-mail address.</td>
</tr>
<tr>
<td>STRING</td>
<td>Represents a character string.</td>
</tr>
<tr>
<td>TIME</td>
<td>Represents an instant of time that recurs every day. When the PDP evaluates time specified in a constraint or obligation, it interprets it as the local time of the host on which the PDP is deployed.</td>
</tr>
<tr>
<td>X500_NAME</td>
<td>Represents a Distinguished Name (DN).</td>
</tr>
<tr>
<td>YEARMONTH_DURATION</td>
<td>Represents a duration of time; can contain years and months.</td>
</tr>
</tbody>
</table>

The data types can be used for the following:

- Declaration of attributes by assigning the data type to an attribute (dynamic attributes, resource attributes, custom attributes retrieved by a custom Attribute Retriever)
- Constraints in policy rules
- Return values in Obligations
- Input parameters
- Return’s declaration of customer/built-in function

Table 2–3 lists the functional categories that can be used with each data type.
<table>
<thead>
<tr>
<th>Function Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARITHMETIC_FUNCTION</td>
<td>Takes two arguments of the integer or double data type and returns an integer or double data type</td>
</tr>
<tr>
<td>BAG_FUNCTION</td>
<td>Operates on a bag (an unordered collection of values in which there may be duplicate values) of primitive data types to decide, for example, what data type or how many values in the bag</td>
</tr>
<tr>
<td>CUSTOM_FUNCTION</td>
<td></td>
</tr>
<tr>
<td>DATE_TIME_ARITHMETIC_FUNCTION</td>
<td>Performs arithmetic operations with date and time data types.</td>
</tr>
<tr>
<td>EQUALITY_COMPARISON_FUNCTION</td>
<td>Compares two arguments (integer or double data types) and yields a boolean as to whether they are equal or not</td>
</tr>
<tr>
<td>GREATER_THAN_COMPARISON_FUNCTION</td>
<td>Compares two arguments (integer or double data types) and yields a boolean as to whether one is greater than the other</td>
</tr>
<tr>
<td>GREATER_THAN_EQUAL_COMPARISON_FUNCTION</td>
<td>Compares two arguments (integer or double data types) and yields a boolean as to whether one is greater than or equal to the other</td>
</tr>
<tr>
<td>HAS_VALUE_EXISTENCE_FUNCTION</td>
<td>This function takes an attribute name as argument. It returns true if the attribute has been assigned a value.</td>
</tr>
<tr>
<td>HIGHER_ORDER_BAG_FUNCTION</td>
<td>Perform operations on bags such that functions may be applied to the bags in general</td>
</tr>
<tr>
<td>LESS_THAN_COMPARISON_FUNCTION</td>
<td>Compares two arguments (integer or double data types) and yields a boolean as to whether one is less than the other</td>
</tr>
<tr>
<td>LESS_THAN_EQUAL_COMPARISON_FUNCTION</td>
<td>Compares two arguments (integer or double data types) and yields a boolean as to whether one is less than or equal to the other</td>
</tr>
<tr>
<td>LOGICAL_FUNCTION</td>
<td>Operates on boolean data type arguments</td>
</tr>
<tr>
<td>NUMERIC_CONVERSION_FUNCTION</td>
<td>Takes one argument of the integer or double data type and converts it to the double or integer data type, respectively</td>
</tr>
<tr>
<td>REGEXP_FUNCTION</td>
<td>Compares two arguments, the first one is a string that represents a regular expression. Checks if the second arguments (in any type) matches the regular expression</td>
</tr>
<tr>
<td>SET_FUNCTION</td>
<td>Operates on two bags mimicking sets (by eliminating duplicate elements)</td>
</tr>
<tr>
<td>SPECIAL_MATCH_FUNCTION</td>
<td>Operates on various data types and evaluate to a boolean based on the specified standard matching algorithm</td>
</tr>
<tr>
<td>STRING_CONVERSION_FUNCTION</td>
<td>Takes a string data type argument and converts it as defined</td>
</tr>
<tr>
<td>STRING_FUNCTION</td>
<td>Takes a string and converts it to other data types</td>
</tr>
<tr>
<td>TYPE_IS_IN_EXISTENCE_FUNCTION</td>
<td>Takes two arguments: the first one is simple type object, and the second is a bag of same type. This function returns true if the value of first argument is present in the bag represented by the second argument.</td>
</tr>
</tbody>
</table>
The functions can be used for the following:

- Constraints in policy rules
- Return values in Obligations
- Input parameters of custom/built-in functions

For more information on the data types and how they might be used, see the XACML 2.0 specifications at [http://docs.oasis-open.org/xacml/](http://docs.oasis-open.org/xacml/).

### 2.3 Executing A Simple Policy

Executing the simple policy procedure documented in Section 1.2, "Composing A Simple Policy" requires that the objects be created in a particular order. For example, a ResourceEntry object can only be created after defining a ResourceType object. The following sections are listed in the correct order for executing a simple policy programmatically.

- Section 2.3.1, "Accessing the Policy Store"
- Section 2.3.2, "Creating an Application Policy"
- Section 2.3.3, "Defining Resource Types"
- Section 2.3.4, "Instantiating a Resource"
- Section 2.3.5, "Associating Actions with the Resource"
- Section 2.3.6, "Specifying a Policy Rule"
- Section 2.3.7, "Specifying the Principal"
- Section 2.3.8, "Defining the Policy"

#### Note:
Before creating any policy objects, you should determine the overall organizational structure of the policy model components; for example, it may be beneficial to implement only one ApplicationPolicy object and, within that parent, delegate policies in multiple PolicyDomainEntry objects. For more information, see Section 5.7, "Delegating with a Policy Domain."

### 2.3.1 Accessing the Policy Store

Any policy management activity must be preceded by retrieving an instance of the PolicyStore object. The following procedure shows how the PolicyStore object is retrieved using interfaces in the oracle.security.jps package. Smith is specified as the user with the administrative rights to manage the policies.

#### Caution:
Errors will be returned for any methods the user is not authorized to call.
1. Retrieve an instance of PolicyStore. The following sample assumes that the code is launched with JVM argument `-Doracle.security.jps.config="absolutepath of jps-config.xml"`.

```java
JpsContextFactory ctxFact = JpsContextFactory.getContextFactory();
JpsContext ctx = ctxFact.getContext();
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
if (ps == null) {
    // if no policy store instance configured in jps-config.xml
    System.out.println("no policy store instance configured");
    return;
}
```

JpsContext declares a collection of service instances common to a particular domain in the file that configures Oracle Platform Security Services, jps-config.xml. If there is more than one JpsContext defined in the jps-config.xml, the policy store specified in the default JpsContext will be returned. You can also get a particular JpsContext by name.


**Notes:**
- If no policy store instance is configured, `getServiceInstance()` will return null.
- If a connection is not established, `getServiceInstance()` throws `javax.persistence.PersistenceException`.
- If the PolicyStore object was already retrieved but the connection went down, any operations within the policy store will lead to a PolicyStoreException. If the backend is down, resume operations once the backup is available. High availability for backend stores is also possible.

2. Specify the administrative user.

```java
Subject smith = new Subject();
Principal principal = new WLSUserImpl("smith");
smith.getPrincipals().add(principal);
BindingPolicyStore ps = BindingPolicyStoreFactory.getInstance();
ps.setSubject(smith);
```

This code is required only if the subject is not set using the application container's authentication mechanism. In the case of the subject being set with the authentication mechanism, the authenticated subject retrieved is used to authorize access to the policy store; it is not required to use BindingPolicyStore.

### 2.3.2 Creating an Application Policy

An ApplicationPolicy object is a container for all objects needed to define secure access to a particular application. An ApplicationPolicy object should be created for each target to be secured. You may create as many as needed. Once created, the ApplicationPolicy is represented by the ApplicationPolicy interface which contains the programmatic managers needed to create resources, policies and other security objects used to define the application's access requirements. These security objects comprise those defined in Chapter 1, "Using the Policy Model."
You can create, delete and retrieve ApplicationPolicy objects with the methods found in the PolicyStore interface. Example 2–1 illustrates how to create an ApplicationPolicy object using the createApplicationPolicy() method.

Example 2–1  Using createApplicationPolicy() Method

ApplicationPolicy ap = ps.createApplicationPolicy("Trading", "Trading Application","Trading Application.");

The values of the createApplicationPolicy() parameters are defined as:

- **Name** - Trading is a unique identifier for the ApplicationPolicy object.
- **Display Name** - Trading Application is an optional, human-readable name for the ApplicationPolicy object.
- **Description** - Trading Application. is optional information describing the ApplicationPolicy object.

**Note:** The ApplicationPolicy object is represented in the Oracle Entitlements Server Administration Console as an Application.

**Caution:** Deleting an ApplicationPolicy object deletes all child objects created within it.

2.3.3 Defining Resource Types

A ResourceTypeEntry object specifies the full scope of traits for a particular kind of resource. From the higher level ResourceTypeEntry object (associated with an ApplicationPolicy object), you instantiate a specific Resource object to represent an actual, secured target. Thus, the ResourceTypeEntry contains a full spectrum of resource attributes (specifications that an instance of this resource type may have) and definitions of all possible valid actions that can be performed on the protected instance. The actions added to a ResourceTypeEntry can be standard actions (GET and POST to a URL) or a custom action on a business object (transfer to or from a bank account).

**Note:** The ResourceTypeEntry object is represented in the Oracle Entitlements Server Administration Console as a Resource Type.

To create, delete, retrieve or modify a ResourceTypeEntry, obtain an instance of the ResourceTypeManager. Example 2–2 creates a ResourceTypeEntry named TradingResType within the TradingApp ApplicationPolicy object. TradingResType has two permissible actions (BUY and SELL) and an attribute named ManagerType.

Example 2–2  Using the createResourceType() Method

ResourceTypeManager resourceTypeManager = tradingApp.getResourceTypeManager();
List<String> actions = new ArrayList();
    actions.add("get");
List<AttributeEntry<? extends DataType>> attributes = new ArrayList<AttributeEntry<? extends DataType>>()
    .add(new BasicAttributeEntry<OpssString>("orderNo", OpssString.class));
    attributes.add(new BasicAttributeEntry<OpssBoolean>("orderNo", OpssString.class));
Executing A Simple Policy

Constructing A Policy Programmatically

```java
(','isDomestic', OpssBoolean.class));
ResourceTypeEntry resType = rtm.createResourceType
('TradingResType', "Trading ResType", "Trading Resource Type",
    actions, attributes, ",", null);
```

tradingApp is the name of the ApplicationPolicy object from which the
ResourceTypeManager is being retrieved. The values of the createResourceType() parameters are defined as:

- **Name** - TradingResType is a unique identifier for the ResourceTypeEntry.
- **Display Name** - Trading ResType is an optional, human-readable name for the ResourceTypeEntry.
- **Description** - Trading Resource Type is optional information describing the ResourceTypeEntry.
- **Actions** - actions is the name of an ordered collection (list) of all valid actions on the ResourceTypeEntry - in this case, GET. Use the setAllAction() method to set a default action keyword.
- **Attributes** - attributes specifies a listing of all valid attributes for the ResourceTypeEntry. If there are duplicate ones, they will be handled as one. To define one or more attributes, create them with the createAttribute() method in the ExtensionManager and reference them here. See Section 2.4.3, "Creating Attribute and Function Definitions" for more information. Attributes can also be null.
- **Delimiter** - / (forward slash) is the default delimiter for the actions. Use the setResourceNameDelimiter() method to set a different delimiter. The delimiter is passed to the method as a ResourceTypeEntry.ResourceNameDelimiter enum.
- **Resource Matcher Class** - null signifies there is no resource permission matcher class used for target and action matching. To set an implementation of the Java Permission class, assign the class name as the matcher class. In this case, the Resource Type is bound with a given permission. A policy defined with the ResourceType is a permission-based policy.

ResourceTypeEntry objects can also be defined as hierarchical by invoking the object’s setHierarchicalResource() method. By passing true to the isHierarchical parameter, the ResourceTypeEntry will be set as hierarchical. A hierarchical ResourceTypeEntry can then be used to instantiate a ResourceEntry in which the following applies:

1. A policy applicable to a ResourceEntry created from a hierarchical ResourceTypeEntry is also applicable to any ResourceEntry objects that are its children.
2. Any attribute defined for a ResourceEntry created from a hierarchical ResourceTypeEntry is inherited by any ResourceEntry objects that are its children.

The isHierarchicalResource() method can be used to determine whether a ResourceTypeEntry has been set as hierarchical.

See Section 2.3.4, "Instantiating a Resource" for information on creating a ResourceEntry instance of a ResourceTypeEntry object.

### 2.3.4 Instantiating a Resource

A ResourceEntry object represents a specific, secured target in a protected application. It can represent software components managed by a container (URLs, EJBs, JSPs) or business objects in an application (reports, transactions, revenue charts).
A ResourceEntry object is defined as an instance of a ResourceTypeEntry object. Be sure the appropriate ResourceTypeEntry is defined before attempting to create a ResourceEntry instance. For more information, see Section 2.3.3, "Defining Resource Types."

To create a ResourceEntry object, obtain an instance of the ResourceManager using the getResourceManager() method in the applicable ApplicationPolicy or PolicyDomainEntry. Following that, use the createResource() method to create the object.

Example 2–3 creates a checking account ResourceEntry. Trading refers to the ApplicationPolicy object from which the ResourceManager is being retrieved.

**Example 2–3 Using createResource() Method**

```java
ResourceManager resMgr = Trading.getResourceManager();
List<AttributeEntry<? extends DataType>> attributes = new ArrayList<AttributeEntry<? extends DataType>>();
attributes.add(new BasicAttributeEntry<OpssString>("orderNo", new OpssString("12345678")));
attributes.add(new BasicAttributeEntry<OpssBoolean>("isDomestic", new OpssBoolean(true)));
ResourceEntry checkingRes = resMgr.createResource("Bob_checking1", "Bob Checking Account", "Checking account.", resType, attributes);
```

The values of the createResource() parameters are defined as:

- **Name** - Bob_checking1 is the unique identifier for the ResourceEntry.
- **Display Name** - Bob Checking Account is an optional, human-readable name for the ResourceEntry.
- **Description** - Checking account. is optional information describing the ResourceEntry.
- **Type** - resType is the ResourceTypeEntry object from which the resource will be instantiated.
- **Attributes** - specifies that there are (optional) resource attributes being configured for this ResourceEntry; in this case, the order number and boolean regarding whether the Resource is domestic or not. To define one or more attributes, create them with the createAttribute() method in the ExtensionManager and reference them here.

Once a ResourceEntry is created, it can be paired with actions in a ResourceActionsEntry or included in a PermissionSetEntry. For more information, see Section 2.3.5, "Associating Actions with the Resource" and Section 2.4.4, "Defining Permission Sets."

---

**Note:** See Oracle Fusion Middleware Administrator's Guide for Oracle Entitlements Server for more information on software components and business objects.

---

**Note:** The ResourceEntry object is represented in the Oracle Entitlements Server Administration Console as a Resource.
2.3.5 Associating Actions with the Resource

A ResourceActionsEntry object associates a Resource instance with a set of actions that can be performed on it. The Resource instance is specified as either a static ResourceEntry or a dynamic ResourceNameExpression.

Note: A ResourceActionsEntry is not a named object that is independently managed. It is just an association.

The following sections have more information.

■ Section 2.3.5.1, "Using a ResourceEntry"
■ Section 2.3.5.2, "Using a ResourceNameExpression"

2.3.5.1 Using a ResourceEntry

The procedure to instantiate a ResourceEntry is explained in Section 2.3.4, "Instantiating a Resource." After instantiating a ResourceEntry, build a ResourceActionsEntry object to define the actions that can be performed on the resource. The set of actions are defined in a list using a subset of the legal actions defined in the Resource's corresponding ResourceTypeEntry. Example 2–4 builds a list that defines the association (resActsList) between the ResourceEntry and its actions using the ResourceActionsEntry interface. This example creates a checking account ResourceEntry and associates the checking account with the ability to read it or modify it.

Example 2–4 Building a ResourceActionsEntry with ResourceEntry

ResourceEntry checkingRes = resMgr.createResource("Bob_checking1", "Bob Checking Account", "Checking account.", resType, null);
List<String> actions = new ArrayList<String>();
   actions.add("read");
   actions.add("write");
List<ResourceActionsEntry> resActsList = new ArrayList<ResourceActionsEntry>(checkingRes, actions);

Bob_checking1 is the ResourceEntry. The List defines the applicable actions for Bob_checking1 that will be governed by this ResourceActionsEntry object: read and write. The allowable actions are culled from the parent ResourceTypeEntry.
2.3.5.2 Using a ResourceNameExpression

Instead of using a ResourceEntry, a ResourceNameExpression can be specified. A ResourceNameExpression contains a defined ResourceTypeEntry and a Java regular expression, expressed as a string. The string is used to match the ResourceEntry instance at runtime. For example, assume the policy data in Table 2–4 has been defined. RAE1 and RAE2 are defined with specific ResourceEntry objects, ResType1 and ResType2. RAE3 is defined with a ResourceNameExpression; during the runtime evaluation of Policy3, http://.* is used to match the ResourceEntry and returns ResType1, the ResourceEntry for an HTTP URL.

Table 2–4 Matching ResourceNameExpression Objects

<table>
<thead>
<tr>
<th>ResourceEntry</th>
<th>ResourceActionsEntry</th>
<th>Policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResType1 (HTTP URL)</td>
<td>RAE1 with ResType1 ResourceEntry <a href="http://www.oracle.com">http://www.oracle.com</a> and action GET</td>
<td>Policy1 with RAE1</td>
</tr>
<tr>
<td>ResType2 (HTTPS URL)</td>
<td>RAE2 with ResType2 ResourceEntry <a href="https://www.oracle.com">https://www.oracle.com</a> and action GET</td>
<td>Policy2 with RAE2</td>
</tr>
<tr>
<td></td>
<td>RAE3 with ResType1 ResourceNameExpression http://*/ and action GET</td>
<td>Policy3 with RAE3</td>
</tr>
</tbody>
</table>

Example 2–5 illustrates how to build a ResourceActionsEntry with a ResourceNameExpression.

Example 2–5 Building a ResourceActionsEntry with ResourceNameExpression

```java
// create one ResourceActionEntry
ResourceNameExpression resExpression = new ResourceNameExpression(resTypeName, resNameExp);
ResourceActionsEntry resActionsEntry = new BasicResourceActionsEntry(resExpression, actions);

List<ResourceActionsEntry> resActionsList = new ArrayList<ResourceActionsEntry>();
resActionsList.add(resActionsEntry);
```

Table 2–5 has examples of the ResourceNameExpression. While any regular expression can be used, the pattern expressions listed in the table are processed faster then regular expressions.

Table 2–5 Examples of ResourceNameExpression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific to Resource Type</td>
<td>To specify any action type, use the keyword specific to the Resource Type.</td>
</tr>
<tr>
<td>.&quot;*&quot;</td>
<td>To specify all resources</td>
</tr>
<tr>
<td>&quot;http.*&quot;</td>
<td>To specify all resources beginning with http.</td>
</tr>
</tbody>
</table>
You may also populate a Permission Set with one or more ResourceActionsEntry objects. See Section 2.4.4, "Defining Permission Sets" for more information.

### 2.3.6 Specifying a Policy Rule

A PolicyRuleEntry defines an **Effect** (and optionally a **Condition**). An Effect specifies the possible outcomes of the policy rule. Effects in Oracle Entitlements Server are **GRANT** or **DENY**. When the policy rule is evaluated (coupled with information regarding a principal and a target ResourceEntry), the rights of the subject in terms of the ResourceEntry are determined. All PolicyEntry objects must contain one (and only one) Policy Rule. See Section 2.3.8, "Defining the Policy" for more information.

**Example 2–6** illustrates how to create a PolicyRuleEntry object named myRule programmatically using the BasicPolicyRuleEntry implementation.

**Example 2–6  Create a PolicyRuleEntry**

```
PolicyRuleEntry myRule = new BasicPolicyRuleEntry
    PolicyRuleEntry.EffectType.GRANT, myCondition);
```

The values of the parameters are defined as:

- **Name** - ReportRule is a unique identifier for the policy rule.
- **Display Name** - Report Policy Rule is an optional, human-readable name for the policy rule.
- **Description** - Rule for Reports policy is optional information describing the policy rule.
- **PolicyRuleEntry.EffectType** - takes a value of GRANT based on the desired outcome. The PolicyRuleEntry.EffectType enum defines the available effect types for Oracle Entitlements Server. The constants are GRANT or DENY.
- **Condition** - myCondition is the name of the optional Condition used by this policy rule. The Condition is a BooleanExpressionEntry which represents an Expression that returns a boolean value. See Section 2.4.5, "Defining a Condition" for more information.

### 2.3.7 Specifying the Principal

A PrincipalEntry specifies the users, groups, or roles to which the policy pertains. It is added to a PolicyEntry object as documented in Section 2.3.8, "Defining the Policy." Table 2–6 illustrates the types of principals and how each can be specified programmatically.
When a policy’s subject is multiple groups and/or roles, that policy applies to a user based on the principal semantic defined. Options include:

- **PRINCIPAL_AND_SEMANTIC** defines a policy that applies to a user if the user matches ALL groups or roles listed as the principal. For example, if a list of principals contains two roles, the user must be member of both roles for the policy to apply.

- **PRINCIPAL_OR_SEMANTIC** defines a policy that applies to a user if the user matches AT LEAST one of the groups or roles listed as the principal. For example, if a list of principals contains two roles, the user can be a member of ONLY one of these roles for the policy to apply.

### 2.3.8 Defining the Policy

A Policy specifies the access rights that specific principals have on specific resources. Basically, it consolidates all the pieces needed to create the access control - including,

<table>
<thead>
<tr>
<th>Principal Type</th>
<th>Example</th>
</tr>
</thead>
</table>
| User             | Specify the class name of the user principal validation provider (weblogic.security.principal.WLSUserImpl) and the user name. The following example defines a user named smith.  
PrincipalEntry aUser = new BasicPrincipalEntry  
("weblogic.security.principal.WLSUserImpl", "smith");  
List<PrincipalEntry> myPrincipal = new  
ArrayList<PrincipalEntry>(); myPrincipal.add(aUser);  

| Group            | Specify the class name of the group principal validation provider (weblogic.security.principal.WLSGroupImpl) and the group name. The following example defines a group named Acme.  
PrincipalEntry aGroup = new BasicPrincipalEntry  
("weblogic.security.principal.WLSGroupImpl", "Acme");  
List<PrincipalEntry> myPrincipal = new  
ArrayList<PrincipalEntry>(); myPrincipal.add(aGroup);  

| Role             | Retrieve the tRole Application Role and add it to the PrincipalEntry.  
AppRoleEntry aRole =  
appRoleManager.getAppRole(tRole);  
List<PrincipalEntry> principal =  
new ArrayList<PrincipalEntry>();  
principals.add(tRole);  

See Section 2.4.1, "Creating Application Roles" for more information.  

| Anonymous Role   | Add anonymous as a principal to policies that allow access to anonymous users.  
PrincipalEntry anonymous = new AnonymousRoleEntry();  
List<PrincipalEntry> principals = new  
ArrayList<PrincipalEntry>();  
principals.add(anonymous);  

See Section 2.4.1, "Creating Application Roles" for more information.  

| Authenticated Role | Add authenticated as a principal to policies that allow access to authenticated users.  
PrincipalEntry authenticated = new AuthenticatedRoleEntry();  
List<PrincipalEntry> principals = new  
ArrayList<PrincipalEntry>();  
principals.add(authenticated);  

See Section 2.4.1, "Creating Application Roles" for more information.
but not limited to, a PolicyRuleEntry, a ResourceActionsEntry, and a PrincipalEntry.

A Policy is programmatically represented as a PolicyEntry object. To create a PolicyEntry object, obtain an instance of the PolicyManager using the getPolicyManager() method. Following that, use the createPolicy() method to create the object. Example 2–7 creates a policy named myPolicy.

**Example 2–7 Using createPolicy() Example**

```java
PolicyManager policyMgr = domain.getPolicyManager();

List<PermissionSetEntry> permSets = new ArrayList<PermissionSetEntry>();
permSets.add(permSet1);
permSets.add(permSet2);

List<PrincipalEntry> principals = new ArrayList<PrincipalEntry>();
principals.add(appRole1);
principals.add(new BasicPrincipalEntry(WLSUserImpl.class.getCanonicalName(),
    "john");

PolicyEntry myPolicy = policyManager.createPolicy
("BankPolicy", "Bank policy", "Policy for bank.", myRule, 
permSets, principals, null, obligations, PolicyEntry.POLICY_SEMANTIC.AND);
```

domain refers to the Policy Domain under which the policy is being created. The values of the createPolicy() parameters are defined as:

- **Name** - Bank Policy is a unique identifier for the PolicyEntry.
- **Display Name** - Bank policy is an optional, human-readable name for the PolicyEntry.
- **Description** - Policy for bank. is optional information describing the PolicyEntry.
- **Policy Rule** - myrule is the PolicyRuleEntry object.
- **PermissionSetEntry** - permSets is a collection (list) of PermissionSetEntry objects. See Section 2.4.4, "Defining Permission Sets" for more information.
- **Principal** - principals is an ordered collection (list) of PrincipalEntry objects defined as the subject of this policy.
- **ResourceActionsEntry** - A list of ResourceActionsEntry objects can also be defined. If the list of PermissionSetEntry objects is null, this list should contain at least one valid element.
- **Obligations** - A list of ObligationEntry objects may be used. See Section 2.4.6, "Adding Obligations" for more information.
- **policySemantic** - describes how principals specified in the policy should be handled. The PolicyEntry.POLICY_SEMANTIC enum defines the available constants as AND or OR.
  - PolicyEntry.POLICY_SEMANTIC.AND applies to a user if the user matches all principals listed in the policy. For example, if a list of principals contains two roles, the user must be a member of both roles for the policy to apply.
  - PolicyEntry.POLICY_SEMANTIC.OR applies to a user if the user matches at least one of the principals listed in the policy. For example, if list of principals contains two roles, the user can be a member of at least one of these roles for the policy to apply.
2.4 Creating Fine Grained Elements for a Simple Policy

Section 2.3, "Executing A Simple Policy" documented how to create the minimum components needed to define a policy. The following sections contain information on how to add the advanced policy elements discussed in Section 1.3, "Adding Fine Grained Objects to a Simple Policy" to a simple policy.

- Section 2.4.1, "Creating Application Roles"
- Section 2.4.2, "Creating Role Mapping Policies"
- Section 2.4.3, "Creating Attribute and Function Definitions"
- Section 2.4.4, "Defining Permission Sets"
- Section 2.4.5, "Defining a Condition"
- Section 2.4.6, "Adding Obligations"

2.4.1 Creating Application Roles

An AppRoleEntry object is associated with an ApplicationPolicy to group access rights that can then be distributed to users who are granted the Application Role. Once an AppRoleEntry is defined, the grantAppRole method can be used to assign the role to a principal statically or a Role Mapping Policy can be created to assign it to principals dynamically. (See Section 2.4.2, "Creating Role Mapping Policies" for more information.) The following can be added as members to an AppRoleEntry:

- Enterprise users from an identity store
- Enterprise roles from an identity store
- Other Application Roles in a policy store

**Note:** The AppRoleEntry object is represented in the Oracle Entitlements Server Administration Console as an Application Role. Application Roles are consolidated under the Role Catalog branch of the Administration Console navigation tree.

When an Application Role is specified as a principal for a particular policy, all users assigned to the role are governed by that policy. All ApplicationPolicy containers have two implicit Application Roles:

- Anonymous Role — implicitly assigned to all unauthenticated users.
- Authenticated Role — implicitly assigned to all authenticated users.

To create an AppRoleEntry, get an instance of AppRoleManager from within the ApplicationPolicy object where the Application Role will be created and use the createAppRole() method. Example 2–8 shows the creation of an AppRoleEntry named TraderRole.

**Example 2–8 Creating an Application Role**

```java
AppRoleManager roleMgr = bankApplication.getAppRoleManager();
AppRoleEntry traderRole = roleMgr.createAppRole("TraderRole",
    "Trader Role", "Trader role");
```

bankApplication defines the ApplicationPolicy object for which we are retrieving the AppRoleManager. The values of the createAppRole() parameters are defined as:
■ Name - TraderRole is a unique identifier for the AppRoleEntry object.
■ Display Name - Trader Role is an optional, human-readable name for the AppRoleEntry object.
■ Description - Trader Role is optional information describing the AppRoleEntry object.

To assign a Principal to an AppRoleEntry object, build a PrincipalEntry list containing the appropriate users or groups. Use grantAppRole() to assign the role to the principals in the list. Example 2–9 shows the creation and assignment of user JSMITH to the TraderRole.

**Example 2–9 Assigning Principals to an Application Role**

```java
//create user named JSMITH PrincipalEntry aUser = new BasicPrincipalEntry("weblogic.security.principal.WLSUserImpl", "JSMITH");
//Add user to principals list
List<PrincipalEntry> principalList = new ArrayList<PrincipalEntry>();
principalList.add(aUser);

//assign user to role.
roleMgr.grantAppRole(traderRole, principalList);
```

The values of the grantAppRole() parameters are defined as:

■ Name - TraderRole is the name of the AppRoleEntry object to which the user is being assigned.
■ Principal - principal is the name of the list which contains the user being added.

Application Role hierarchies can be built by assigning Application Roles as members of other Application Roles. A policy that applies to an Application Role also applies to all Application Roles that have been assigned to it as members. Example 2–10 illustrates how the TraderManagers role is assigned as a member of the AllManagers role. Thus, all policies that apply to members of the AllManagers role also apply to all members of the TraderManagers role.

**Example 2–10 Applying Application Role Hierarchies**

```java
//create AllManagers and TraderManagers roles
AppRoleEntry allManagers = roleMgr.createAppRole("AllManagers", "AllManagers Role", "Role for all managers.");
AppRoleEntry traderManagers = roleMgr.createAppRole("TraderManagers", "TraderManagers Role", "Role for Trader managers.");

//add TraderManagers to a principals list
List<PrincipalEntry> principalList = new ArrayList<PrincipalEntry>();
principalList.add(traderManagers);

//add TraderManagers role as principal of AllManagers role
roleMgr.grantAppRole(allManagers, principalList);
```

### 2.4.2 Creating Role Mapping Policies

A Role Mapping Policy is created at the ApplicationPolicy level - the same level at which the Application Role is defined. A RolePolicyEntry object represents a Role Mapping Policy. It provides the methods to define a policy that will determine if a user or group is granted or denied an Application Role.
Creating Fine Grained Elements for a Simple Policy

To create a RolePolicyEntry object, obtain an instance of the RolePolicyManager using the getRolePolicyManager() method in the applicable ApplicationPolicy. Following that, use the createRolePolicy() method to create the object.

**Example 2–11 Using the createRolePolicy() Method**

```java
//get the RolePolicyManager
RolePolicyManager roleMapPolicyManager = TellerApp.getRolePolicyManager();

List<AppRoleEntry> appRoles = new ArrayList<AppRoleEntry>();
appRoles.add(appRole1);

List<PrincipalEntry> principalList = new ArrayList<PrincipalEntry>();
principals.add(new BasicPrincipalEntry(WLSUserImpl.class.getCanonicalName(),
"john");

PolicyRuleEntry rule = new BasicRuleEntry("rule", "rule for role policy", "rule for role policy", EffectType.GRANT, null);

List<ResourceEntry> resources = new ArrayList<ResourceEntry>();
resources.add(resourcel);

//create the RolePolicyEntry
RolePolicyEntry rolepolicy = roleMapPolicyManager.createRolePolicy
("TellerRoleMapping", "Teller Role Mapping", "Teller Role Mapping Policy",
appRoles, principals, rule, resources, null);

TellerApp is the name of the ApplicationPolicy object from which the
RolePolicyManager is being retrieved. The values of the createRolePolicyEntry() parameters are defined as:

- **Name** - TellerRoleMapping is a unique identifier for the RolePolicyEntry.
- **Display Name** - Teller Role Mapping is an optional, human-readable name for the RolePolicyEntry.
- **Description** - Teller Role Mapping Policy is optional information describing the ResourceTypeEntry.
- **Application Roles List** - appRoles is an ordered collection (list) of all application roles to grant (or deny) on evaluation of the RolePolicyEntry.
- **Principals List** - principals is a collection (list) of PrincipalEntry objects to map to the Application Roles. This value cannot be an ApplicationRole or an Administration Role, and the list cannot be empty.

**Note:** Role Mapping Policies use only the OR semantic. See Section 2.3.7, "Specifying the Principal" for more information.

- **Policy Rule** - rule is the PolicyRuleEntry object that defines a Condition for the Role Mapping Policy. A value is required.
2.4.3 Creating Attribute and Function Definitions

An attribute or function definition is metadata that describes a specific attribute or function. Among other information, it defines the name of the attribute or function, the type of data the attribute takes, or the function returns, as a value and whether said value is single or multiple. The metadata informs Oracle Entitlements Server how to deal with the particular attribute or function that is being defined.

Attribute and function definitions can be used in a Condition or an Obligation. In regards to a Condition, attribute and function definitions can be used to make an optional expression that can be added to a policy to further restrict access to the protected resource. In regards to an Obligation, this optional set of name-value pairs returns additional information, with a policy decision, to the Policy Enforcement Point (PEP). There are two ways to define an Obligation:

- Statically where an attribute with an absolute value is returned.
- Dynamically where an attribute value, or a custom function, is evaluated at runtime and the output is returned.

Note: See Section 1.3.3, "Adding a Condition" and Section 1.3.5, "Building an Obligation" for more general information. Section 2.4.5, "Defining a Condition" and Section 2.4.6, "Adding Obligations" contain additional coding information.

Attribute and function definitions are managed at the ApplicationPolicy level. You can use definitions pre-defined for Oracle Entitlements Server or, you can define new ones to suit your requirements using the ExtensionManager. Information on creating custom definitions is in the following sections.

- Section 2.4.3.1, "Creating Attribute Definitions."
- Section 2.4.3.2, "Creating Custom Function Definitions."

Information on the pre-defined RuleExpressionEntry.BuiltInAttributes and RuleExpressionEntry.BuiltInFunctions enum definitions can be found in the Oracle Fusion Middleware Management Java API Reference for Oracle Entitlements Server.

2.4.3.1 Creating Attribute Definitions

An AttributeEntry object can be a value dynamically defined at runtime (for example, the locality of the user) or a value based on the type of protected resource (for example, creation date of a text file). During policy evaluation, attribute values can
be passed in by the application or Oracle Entitlements Server can retrieve it using a custom attribute retriever.

---

**Note:** Dynamic attribute definitions are managed as a child object of the ApplicationPolicy so that they may be used in policies within different Policy Domains. See Chapter 5, "Delegating Policy Administration" for information on Policy Domains.

---

To create an attribute definition, get an instance of the ExtensionManager and use the createAttribute() method. Example 2–12 creates an attribute definition named myAttr.

**Example 2–12 Creating a Dynamic Attribute Definition**

```java
//get the ExtensionManager
ExtensionManager xMgr = bankApplication.getExtensionManager();

//create the dynamic attribute
AttributeEntry<OpssString> attr = xMgr.createAttribute
    ("min_age", "minimum age", "minimum age of subject.", OpssString.class,
    AttributeEntry.AttributeCategory.DYNAMIC, true);
```

`bankApplication` refers to the ApplicationPolicy object under which the extension is being created. The values of the createAttribute() parameters are defined as:

- **Name** - `min_age` is a unique identifier for the attribute.
- **Display Name** - `minimum age` is an optional, human-readable name for the attribute.
- **Description** - `minimum age of subject.` is optional information describing the attribute.
- **Data Type** - `OpssString.class` is the attribute's data type; in this case, a string. This parameter takes a value of any of the sub classes of the `oracle.security.jps.service.policystore.info.DataType` class.

---

**Note:** `attr.setValue(new OpssString("John"))` is a line of code that would set the value of the string as John.

---

- **Category** - `AttributeEntry.AttributeCategory.DYNAMIC` defines the attribute as dynamic. This can be DYNAMIC or RESOURCE. The value of a dynamic attribute is passed with the authorization request or retrieved by the Policy Decision Point. The value of a resource attribute is defined by the resource instance.
- **isSingleValue** - true indicates that the attribute takes a single value. A value of false would indicate multiple values.

### 2.4.3.2 Creating Custom Function Definitions

A custom function represents some externally implemented logic that is used to generate an output which is returned to the PDP; the value is then used in a Condition. Example 2–13 illustrates how to create a custom function by retrieving the ApplicationPolicy under which the function will be created and getting an instance of the ExtensionManager.
Example 2–13 Creating a Custom Function Definition

```java
ApplicationPolicy ap = ps.getApplicationPolicy("MyAppPolicy");
ExtensionManager xMgr = ap.getExtensionManager();
FunctionEntry func = xMgr.createFunction("myFunc",
    "Credit Standing Function", "Returns credit standing.",
    "acme.demo.CreditStanding", OpssBoolean.class, params);
```

MyAppPolicy is the identifier for the ApplicationPolicy object under which the function is being created. The values of the createFunction() method parameters are defined as:

- **Name** - myFunc is a unique identifier for the FunctionEntry.
- **Display Name** - Credit Standing Function is an optional, human-readable name for the FunctionEntry.
- **Description** - Returns credit standing. is optional information describing the FunctionEntry.
- **Class Name** - acme.demo.CreditStanding is the fully-qualified name of the class implementing the FunctionEntry.
- **Return Data Type** - Any sub class of the `oracle.security.jps.service.policystore.info.DataType` class which is a super class comprised of all data types supported by the policy store (OpssBoolean, OpssDate, OpssInteger, OpssString, OpssTime).
- **Input Data Type** - params denotes the input data type for the function. It is one of the sub classes of the `oracle.security.jps.service.policystore.info.DataType` class which is a super class comprised of all data types supported by the policy store (OpssBoolean, OpssDate, OpssInteger, OpssString, OpssTime).

For more information, see Section 2.4.5, "Defining a Condition" and Section 7.2, "Developing Custom Functions."

### 2.4.4 Defining Permission Sets

As documented in Section 1.2, "Composing A Simple Policy," a PermissionSetEntry object is used to aggregate one or more ResourceActionsEntry objects. A ResourceActionsEntry object is a pairing of the resource being secured with the action(s) that the policy will allow or deny on it. (See Section 2.3.5, "Associating Actions with the Resource" for more information on ResourceActionsEntry objects.) With the PermissionSetEntry, you can bundle ResourceActionsEntry objects as needed. If a resource is referenced in the ResourceActionsEntry object, the resource must be defined in the same policy domain as the permissionSet is defined (or modified) - the later is applicable when the permissionSet is modified when a resource is added to it. This is a construct that can be used instead of the standard RBAC role aggregations.

**Note:** The PermissionSetEntry object is represented in the Oracle Entitlements Server Administration Console as an Entitlement.

Example 2–14 illustrates how to create a PermissionSetEntry object. It includes the code for creating a ResourceEntry and ResourceActionsEntry. *domain* is the name of the Policy Domain from which the instance of the PermissionSetManager is retrieved.
Example 2–14  Building a PermissionSetEntry

//get the PermissionSetManager
PermissionSetManager psMgr = domain.getPermissionSetManager();

//create a ResourceEntry and ResourceActionsEntry
ResourceManager resMgr = domain.getResourceManager();
ResourceEntry checkingRes = resMgr.createResource("Bob_checking1", "Bob Checking Account", "Checking account.", type, null);
List<String> actions = new ArrayList<String>;
actions.add("read");
actions.add("write");
List<ResourceActionsEntry> resActsList = new
ArrayList<ResourceActionsEntry>();
resActsList.add(new BasicResourceActionsEntry(checkingRes, actions));

//create a PermissionSetEntry
PermissionSetEntry permSet =
permSetManager.createPermissionSet("RptsPermSet", "Reports Permission Set", "Permission set for Reports policy.", resActsList);

The values of the createPermissionSet() parameters are defined as:

- **Name** - RptsPermSet is a unique identifier for the PermissionSetEntry object.
- **Display Name** - Reports Permission Set is an optional, human-readable name for the PermissionSetEntry object.
- **Description** - Permission set for Report policy. is optional information describing the PermissionSetEntry object.
- **ResourceActionsEntry** - resActsList is the ResourceActionsEntry being associated with this PermissionSetEntry object.

2.4.5 Defining a Condition

An optional Condition in a policy rule can be used to set additional requirements on a decision returned in response to a request for access. For example, a Condition can be used to grant access to a resource only on the condition that the request was issued from a specific location or at a specific time. A Condition is written in the form of an expression that resolves to either true or false. If the expression resolves to true, the condition is satisfied and the policy is applicable. If the expression does not resolve to true, the policy is not applicable.

**Note:** Conditions in Role Mapping Policies provide the same functionality, and take the same format, as Conditions in Authorization Policies.

A Condition is defined in a PolicyRuleEntry as discussed in Section 2.3.6, "Specifying a Policy Rule." It is an expression built using attributes or functions that can (optionally) be added to the policy rule to further restrict it. The expression is evaluated using dynamic or resource attribute values, or values returned from component functions.

A Condition must return true or false so the expression can only return true or false; thus, it must be defined in a BooleanExpressionEntry. The BooleanExpressionEntry may:

- Have an unlimited number of ExpressionComponent objects.
An expression object has a function and one or more arguments of the type ExpressionComponent. The ExpressionComponent interface represents any entity that can appear as part of the expression.

**Note:** the order in which components are added to an expression must be the same order in which the parameters appear in the input parameter list. For example, if a function needs (OpssString, OpssTime, OpssInteger), the expression must be constructed as:

```java
ex.addExpressionComponent(<string param>);
ex.addExpressionComponent(<time param>);
ex.addExpressionComponent(<integer param>);
```

The following objects are of the type ExpressionComponent:

- Any DataType object
  
  See Section 2.1, "Using the Java API."

- AttributeEntry

- ValueCollection

- Expression

  - Nest ExpressionComponent objects.

  - Use predefined or custom functions with boolean or non-boolean return types.

- Use predefined or dynamic attributes as function input:

  - A dynamic attribute is one whose value is obtained at evaluation time.

  - A predefined attribute is one whose value is not related to the subject, resource, action of the policy or rule; for example, the time of day.

  - A literal value (defined as an ExpressionComponent) that is of any currently supported data type: Boolean, Date, Integer, String and Time.

- Compare the boolean values returned from two or more expressions using the AND or OR operators.

Example 2–15 illustrates how to define a Condition using the BooleanExpressionEntry class to specify the expression and (optional) parameters.

**Example 2–15 Defining a BooleanExpressionEntry**

```java
BooleanExpressionEntry bexp =
    new BooleanExpressionEntry(expression)
```

The BooleanExpressionEntry parameter has:

- A FunctionEntry for a built-in function, or a custom function obtained using the ExtensionManager.

- Zero or more ExpressionComponent objects. An ExpressionComponent is an interface implemented by Class<? extends DataType>, ValueCollection, AttributeEntry and Expression. The following objects can be used to build an Expression: OpssBoolean, OpssDate, OpssInteger, OpssString, OpssTime, ValueCollection, all classes that implement the AttributeEntry interface, or an Expression itself (nesting). It represents a simple condition such as string1 = string2 or a more complex condition such as `(((checking_balance + savings_
balance) > 10000) AND (customFunc_checkCustomerType(user_name, "GOLD")).

From a high level, a developer must take the following steps to define a Condition as a BooleanExpressionEntry. This procedure assumes the logic detailing the process has been defined; in this example, assume a banking policy is applicable only to users who are GOLD members with a combined savings and checking balance of $10,000.

1. Isolate the individual components of the logic for which AttributeEntry objects will be defined; in this example, an attribute that defines a combined savings and checking balance (to compare with $10,000) and one that defines the type of customer (to compare with GOLD).

2. Identify functions implicit in each component for which FunctionEntry objects will be defined; in this example, there is one function that creates a combined balance (saving_balance + checking_balance > 10000) and one that checks for the customer type (customFunc_checkCustomerType(username, "GOLD")).

3. Build ExpressionComponent objects one by one, identifying them as functions and parameters; in this example, expressions are nested and use the AND operator.

   - integer_add(saving_balance, check_balance)
   - integer_greater_than(integer_add (saving_balance, checking_balance), 10000)
   - customFunc_checkCustomerType(username, "GOLD")
   - and(integer_greater_than(integer_add (saving_balance, checking_balance), 10000, customFunc_checkCustomerType(username, "GOLD"))

4. Build the BooleanExpressionEntry using the ExpressionComponent objects. The preferred way to generate a boolean expression is illustrated in Example 2–16.

   **Example 2–16  Building a BooleanExpressionEntry**

   ```java
   //Define the checking and savings balances and compute one total
   Expression addBalance = new Expression(function entry for integer_add);
   addBalance.add(attribute entry for savings_balance);
   addBalance.add(attribute entry for checking_balance);

   //Compare the total balance to 10,000
   Expression greaterThan = new Expression
   (function entry for integer_greater_than);
   greaterThan.addExpressionComponent(addBalance);
   greaterThan.addExpressionComponent(new OpssInteger(10000));

   //Define the function to check the customer type
   Expression goldMember = new Expression(function entry for customFunc_checkCustomerType);
   goldMember.addExpressionComponent(attribute entry for username);
   goldMember.addExpressionComponent(new OpssString("GOLD"));

   //Compare the outcome using AND operator
   Expression top = new Expression(function entry for AND);
   top.addExpressionComponent(greaterThan);
   top.addExpressionComponent(goldMember);
   ```
The expression constructor is provided with the function entry, and each function argument is added as an expression component from left to right.

---

**Note:** To add all `ExpressionComponent` objects at once, use the `setExpressionComponent(List<ExpressionComponent>)` interface. The list of components must be built in order of the arguments passed to the function; for example, the first component in the list is the first argument passed to the function, the second component is the second argument and so on.

---

5. Create a `BooleanExpressionEntry`.

Oracle Entitlements Server supports many predefined functions to be used in conditions (AND/OR, boolean functions, or string functions). The following sections contain information on the kinds of expressions that can be used.

- Section 2.4.5.1, "Constructing a Boolean Expression"
- Section 2.4.5.2, "Constructing a Custom Function Expression"

### 2.4.5.1 Constructing a Boolean Expression

A boolean expression can evaluate an outcome based on the comparison between two boolean results. The outcome of the comparison would be true or false. A boolean expression allows a policy condition to be based on the results of two or more basic expressions of different value types.

The following code contains two basic expressions and a boolean expression. The integer expression (comparing two integers) and the string expression (comparing two strings) are basic expressions. The boolean expression compares the results returned by the basic expressions.

```java
Expression leftExpression =
    new Expression(function-entry-for-INTEGER_LESS_THAN);
leftExpression.add(attribute entry for userBudget);
leftExpression.add(new OpssInteger(2000));

Expression rightExpression =
    new Expression(function-entry-for-STRING_EQUAL);
rightExpression.addExpressionComponent(thisMonth);
rightExpression.addExpressionComponent(new OpssString("December"));

Expression expression = new Expression(function-entry-for-AND);
expression.addExpressionComponent(leftExpression);
expression.addExpressionComponent(rightExpression);

//boolean expression
RuleExpressionEntry<OpssBoolean> condition =
    new BooleanExpressionEntry<OpssBoolean>(expression);
```

The values of the parameters are defined as:

- `userBudget` - a dynamic attribute that represents a dollar amount
- `2000` - a constant integer
- `function-entry-for-INTEGER_LESS_THAN` - takes a `FunctionEntry` obtained by using the enum
  `(ExtensionManager.getFunctionEntry(BuiltInFunctions.INTEGER_LESS_THAN))`
thisMonth - a dynamic attribute representing the current month

December - a constant string

function-entry-for-STRING_EQUAL - takes a FunctionEntry obtained by using the enum (ExtensionManager.getFunctionEntry(BuiltInFunctions.STRING_EQUAL)

leftExpression / rightExpression - dynamic attributes representing the results of the basic expressions.

December - a constant string

function-entry-for-AND - takes a FunctionEntry obtained by using the enum (ExtensionManager.getFunctionEntry(BuiltInFunctions.AND)

Note: AND returns true only if the results of the basic expressions were also true. The other supported operations for a boolean expression are NOT (takes a single true/false value and negates it) and OR (takes two true/false values and produces one true result if either operand is true).

### 2.4.5.2 Constructing a Custom Function Expression

A custom function expression invokes a custom function and returns true or false based on the outcome. The custom function expression can also include one or more parameters. Once the function is called and any parameter(s) are defined, construct a RuleExpressionEntry object to invoke the function using the parameter(s) as input.

The following code determines whether the client from which the request is being made would be considered low risk. The function analyzes the client type and returns the string *Low Risk* if it is.

```java
//get the ClientType custom function
FunctionEntry function = xMgr.getFunction("ClientType");
Expression ex = new Expression(function);

//add component referencing "LowRisk" string to expression
ex.addExpressionComponent(new OpssString("LowRisk");

//construct BooleanExpressionEntry to invoke function
RuleExpressionEntry<OpssBoolean> = new BooleanExpressionEntry(ex);
```

This second example shows how to build a custom function expression that takes parameters of different expression value types.

```java
// define the acceptable expression value types
List<? extends DataType>> inputParams =
    new ArrayList<Class<? extends DataType>>();
    inputParams.add(OpssInteger.class);
    inputParams.add(OpssString.class);
    inputParams.add(OpssTime.class);

// declare the function
FunctionEntry func = extensionManager.createFunction("ReportsPolicyCondition",
    "ReportsPolicyCondition", "Condition for Reports policy",
    "oracle.demo.oes.ComplexFunction", OpssBoolean.class, inputParams);

// use the function to construct a condition
AttributeEntry<OpssInteger> attrEntry =
extMngr.getAttribute(BuiltinAttributes.SYS_OBJ.toString());

Expression expression = new Expression(func)
expression.addExpressionComponent(new OpssInteger(100));
expression.addExpressionComponent(attrEntry);
expression.addExpressionComponent(new OpssTime(17, 0, 0));

RuleExpressionEntry<OpssBoolean> condition =
    new BooleanExpressionEntry<OpssBoolean>(expression);

---

Note: Custom function expressions do not use comparison operators.

2.4.6 Adding Obligations

An Obligation specifies optional information that is taken into account during policy enforcement. This information is returned to the entity calling for an authorization decision with the resolved effect (GRANT or DENY) and imposes an additional requirement on the policy outcome; for example, if a certain amount of money is withdrawn from a checking account, send a text message to the account holder’s registered mobile phone.

An Obligation is managed as a named object that contains a set of name-value pairs. The object is always managed in the context of a policy. There are two ways to define an Obligation:

■ Statically where an attribute with an absolute value is returned as an Obligation.
■ Dynamically where an attribute value, or a custom function, is evaluated at runtime and the output is returned as the Obligation.

If a policy contains an Obligation, the information is returned to the application as a named ObligationEntry object containing a set of attributes. To specify an Obligation, build an ObligationEntry object that contains the data to return. The following procedure constructs an ObligationEntry that provides the string message Trader managers may run reports.

1. Define the message string using the AttributeAssignment class and add it to an attribute array list named traderRptList.

   ```java
   AttributeAssignment<OpssString> traderRpt = new AttributeAssignment<OpssString>
       ("traderRptMessage", new OpssString("Trader managers may run reports."));
   List<AttributeAssignment<? extends DataType>> traderRptList =
       new ArrayList<AttributeAssignment<? extends DataType>>();
   traderRptList.add(traderRpt);
   ```

   The values of the parameters are defined as:

   ■ Name - traderRptMessage is a unique identifier for the string.
   ■ OpssString - Trader managers may run reports. is the string.

2. Construct the traderRptObl Obligation and traderRptOblList array using the ObligationEntry interface.

   ```java
   ObligationEntry traderRptObl = new BasicObligationEntry
       ("traderRptObl", "Trader Report Obligation",
        "obligation for Trader Report policy.", traderRptList);
   List<ObligationEntry> traderRptOblList = new ArrayList<ObligationEntry>();
   ```
traderRptOblList.add(traderRptObl);

The values of the parameters are defined as:

- **Name** - traderRptObl is a unique identifier for the Obligation.
- **Display Name** - Trader Report Obligation is an optional, human-readable name for the Obligation.
- **Description** - Obligation for Trader Report policy is an optional description of the Obligation.
- **Assignments** - traderRptList is the attribute array list previously created.

3. Specify the obligation when creating the policy.

```java
PolicyEntry policyEntry = policyManager.createPolicy
    ("TraderRpt", "TraderRpt", "Trader report policy.", traderRptRule, traderRptPermissionSetEntryList, traderRptPrincipals, traderRptOblList);
```

The values of the parameters are defined as:

- **Name** - TraderRpt is a unique identifier for the policy.
- **Display Name** - TraderRpt is an optional, human-readable name for the policy.
- **Description** - Trader Report policy is an optional description.
- **Rule** - traderRptRule is the name of the PolicyRuleEntry object.
- **PermSets** - traderRpt is a list of PermissionSetEntry objects.
- **Principals** - traderRptPrincipals is a list of PrincipalEntry objects.
- **Obligations** - traderRptRule is a list of ObligationEntry objects.

---

**Note:** If an application uses an Obligation, it must be requested in the isAccessAllowed() authorization request.

## 2.5 Accessing Code Examples

http://www.oracle.com/technetwork/middleware/oes/overview/index.html contains links from which developer and administration examples for Oracle Entitlements Server can be accessed.
Managing Policy Objects Programmatically

Many of the application programming interfaces (API) documented in Chapter 2, "Constructing A Policy Programmatically" contain methods that allow for managing policy objects programmatically. This chapter contains information on how to use those methods. It contains the following sections.

- Section 3.1, "Managing Policies Using Oracle Entitlements Server"
- Section 3.2, "Using Scope Levels to Manage Policy Objects"

3.1 Managing Policies Using Oracle Entitlements Server

Oracle Entitlements Server allows administrators to perform create, read, update, and delete (CRUD) operations on all policy and global objects. This can be done in any of the following ways:

- Using the Management API (as described in Section 3.2, "Using Scope Levels to Manage Policy Objects")
- Using the Administration Console (as described in Oracle Fusion Middleware Administrator’s Guide for Oracle Entitlements Server)
- Using the WebLogic Scripting Tool on the command line (as described in Oracle Fusion Middleware Administrator’s Guide for Oracle Entitlements Server)

3.2 Using Scope Levels to Manage Policy Objects

The policy store contains three scoping levels under which policies are managed: the top-level Policy Store itself, the Application (Application Policy), and the Policy Domain.

- A PolicyStore object represents the entire policy store. Application policies and system administration policies are managed at this scope. Any policy management activity must be preceded by retrieving an instance of the PolicyStore object as documented in Section 2.3.1, "Accessing the Policy Store." The policy store location, the account and the account password used to access it are defined in jps-config.xml, the Oracle Platform Security Services configuration file. Example 3–1 illustrates how this information is defined in jps-config.xml during installation.

Example 3–1  Definition of a Policy Store in jps-config.xml

  <serviceProviders>
Using Scope Levels to Manage Policy Objects

For more information, see Section 3.2.1, "Managing Objects Created at the PolicyStore Scope."

- An ApplicationPolicy object represents an application being secured by Oracle Entitlements Server. Within an ApplicationPolicy, programmatic objects used to define policies (ResourceTypeEntry, FunctionEntry, AppRoleEntry, RolePolicyEntry and the like) are managed.

Note: Optionally, these programmatic objects can also be managed by creating one (or multiple) PolicyDomainEntry objects within the ApplicationPolicy as described in the following bullet point and in Chapter 5, "Delegating Policy Administration."

For more information, see Section 3.2.2, "Managing Objects Within the ApplicationPolicy Scope."

- An optional PolicyDomainEntry object can be created to partition, and serve as a management point for policy objects and completed policy definitions. One PolicyDomainEntry can be used to maintain all policies securing an application or multiples can be used to organize policy components as needed. Policies are defined using objects created in its parent ApplicationPolicy object. Policy Domains are invisible to each other, even those in a parent-child relationships. Thus, the Resources, Permission Sets and Policies managed in a Policy Domain...
can only be used in that Policy Domain. More information on the Policy Domain can be found in Chapter 5, "Delegating Policy Administration."

For more information, see Section 3.2.3, "Managing Objects within the PolicyDomainEntry Scope."

Administration Roles are managed at all scope levels depending on where they were created. For information on creating and managing Administration Roles, see Chapter 5, "Delegating Policy Administration."

The following sections contain more specific information.

■ Section 3.2.1, "Managing Objects Created at the PolicyStore Scope"
■ Section 3.2.2, "Managing Objects Within the ApplicationPolicy Scope"
■ Section 3.2.3, "Managing Objects within the PolicyDomainEntry Scope"

### 3.2.1 Managing Objects Created at the PolicyStore Scope

Within the PolicyStore object, policy components securing different applications are organized within one or more second level ApplicationPolicy objects. Section 2.3.2, "Creating an Application Policy" documented how to create an ApplicationPolicy object. You can also delete and retrieve ApplicationPolicy objects with the methods found in the PolicyStore interface.

**Note:** The ApplicationPolicy object is represented in the Oracle Entitlements Server Administration Console as an Application.

Example 3–2 illustrates how to delete an ApplicationPolicy object named Trading using the \(\text{deleteApplicationPolicy()}\) method.

**Example 3–2 Using deleteApplicationPolicy() Method**

```java
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ApplicationPolicy ap = ps.deleteApplicationPolicy("Trading");
```

The value of the \(\text{deleteApplicationPolicy()}\) parameter is Trading, the unique identifier defined as the Name when the object was initially created. The \(\text{getApplicationPolicy()}\) method will retrieve the ApplicationPolicy object using the same Name value. Additionally, you can retrieve many ApplicationPolicy objects by calling the \(\text{getApplicationPolicies()}\) method and passing search criteria to it using the ApplicationPolicySearchQuery class.

**Caution:** Deleting an ApplicationPolicy object deletes all child objects created within it.

### 3.2.2 Managing Objects Within the ApplicationPolicy Scope

Within the ApplicationPolicy object, policy components are organized within one or more PolicyDomainEntry objects. Other components managed at the ApplicationPolicy level include Resource Types, Application Roles, Role Policies and Extensions (Functions and Attributes). The following sections have more information.

■ Section 3.2.2.1, "Managing PolicyDomainEntry Objects"
■ Section 3.2.2.2, "Managing ResourceTypeEntry Objects"
Using Scope Levels to Manage Policy Objects

- Section 3.2.2.3, "Managing and Granting AppRoleEntry Objects"
- Section 3.2.2.4, "Managing Role Mapping Policy (RolePolicyEntry) Objects"
- Section 3.2.2.5, "Managing AttributeEntry and FunctionEntry Objects"
- Section 3.2.2.6, "Managing ResourceEntry Objects"
- Section 3.2.2.7, "Managing Permission Sets"
- Section 3.2.2.8, "Managing the Policy"

3.2.2.1 Managing PolicyDomainEntry Objects

Section 5.7, "Delegating with a Policy Domain" documents how to create an optional PolicyDomainEntry object that can be used to help partition policy definition components. You can also delete and retrieve PolicyDomainEntry objects with the methods found in the ApplicationPolicy interface. To manage a Policy Domain, obtain an instance of the PolicyDomainManager and call the appropriate method. Example 3–3 illustrates how to delete a PolicyDomainEntry created within the Trading ApplicationPolicy. mydomain is the unique identifier defined as the Name when the object was initially created.

Example 3–3 Using deletePolicyDomain() Method

```java
PolicyDomainManager domainMgr = app.getPolicyDomainManager();
PolicyDomainEntry pdEntry = domainMgr.deletePolicyDomain("mydomain");
```

Example 3–4 illustrates how to modify the Display Name and Description of the PolicyDomainEntry using the setDescription() and setDisplayName() methods available through that interface.

Example 3–4 Using modifyPolicyDomain() Method

```java
PolicyDomainManager domainMgr = app.getPolicyDomainManager();
PolicyDomainManager domainMgr = app.getPolicyDomainManager();
PolicyDomainEntry pdEntry = domainMgr.getPolicyDomain("mydomain");

// modify PolicyDomainEntry displayName and description
pdEntry.setDescription("This is description.");
pdEntry.setDisplayName("Domain Display Name");

// persist the change
domainMgr.modifyPolicyDomain(pdEntry);
```

Example 3–5 illustrates how to retrieve a PolicyDomainEntry using mydomain, the unique identifier defined as the Name when the object was initially created.

Example 3–5 Using getPolicyDomain() Method

```java
PolicyDomainManager domainMgr = app.getPolicyDomainManager();
PolicyDomainEntry PDEntry = domainMgr.getPolicyDomain("mydomain");
```

Additionally, you can retrieve many PolicyDomainEntry objects by calling the getPolicyDomains() method and passing search criteria to it using the PolicyDomainSearchQuery class.
3.2.2.2 Managing ResourceTypeEntry Objects

Section 2.3.3, "Defining Resource Types" documented how to create a ResourceTypeEntry object. You can also delete, modify and retrieve ResourceTypeEntry objects by getting an instance of the ResourceTypeManager (using getResourceTypeManager() in the ApplicationPolicy interface) and calling the appropriate method.

Note: The ResourceTypeEntry object is represented in the Oracle Entitlements Server Administration Console as a Resource Type.

Example 3–6 deletes a ResourceTypeEntry named TradingResType within the Trading ApplicationPolicy object.

Example 3–6 Using the deleteResourceType() Method

```java
//get the ResourceTypeManager
ResourceTypeManager resourceTypeManager = Trading.getResourceTypeManager();

//delete the Resource Type
resourceTypeManager.deleteResourceType("TradingResType", "true");
```

Trading is the name of the ApplicationPolicy under which the ResourceType object was created. TradingResType is the name of the ResourceType object being deleted.

The values of the deleteResourceType() parameters are defined as:

- **Name**: TradingResType is the unique identifier defined as the Name when the object was initially created.
- **cascadeDelete**: This parameter takes a value of true or false and governs how the ResourceTypeEntry and related objects would be removed. If true, the ResourceTypeEntry and all instantiated ResourceEntry objects are deleted. If false, and ResourceEntry instances exist, the operation fails and the PolicyStoreOperationNotAllowedException is thrown.

The getResourceType() method can be used to retrieve a ResourceTypeEntry, also by Name. You can retrieve many ResourceTypeEntry objects by calling the getResourceTypes() method and passing search criteria to it using the ResourceTypeSearchQuery class.

3.2.2.3 Managing and Granting AppRoleEntry Objects

Section 2.4.1, "Creating Application Roles" documents how to create an AppRoleEntry object and assign users to it. (When the AppRoleEntry object is then specified as a principal for a particular policy, all users assigned to the role are governed by that policy.) You can also delete, modify and retrieve AppRoleEntry objects by getting an instance of the AppRoleManager (using getAppRoleManager() in the ApplicationPolicy interface) and calling the appropriate method.

Note: The AppRoleEntry object is represented in the Oracle Entitlements Server Administration Console as an Application Role. Application Roles are searched for, and consolidated, under the Role Catalog branch of the Administration Console navigation tree. A Role Catalog is a user interface grouping of all activities related to managing Application Roles and its characteristics. A Role Category is a tag you can assign to a role for ease of management.
Example 3–7 removes an AppRoleEntry named TradingAppRole from the policy store. TradingApp is the name of the ApplicationPolicy under which the AppRoleEntry object was created.

**Example 3–7 Using deleteAppRole() Method**

```
//get the AppRoleManager
AppRoleManager appRoleManager = Trading.getAppRoleManager();

//delete the AppRoleEntry
appRoleManager.deleteAppRole("TradingAppRole", "true");
```

The values of the deleteAppRole() parameters are defined as:

- **Name** - TradingAppRole is the unique identifier defined as the Name when the object was initially created.
- **cascadeDelete** - This parameter takes a value of true or false and governs how the AppRoleEntry and related objects would be removed. If true, the AppRoleEntry is deleted and removed from all policies referencing it. (If it is the only role referenced by a policy, the policy is also removed.) If false, and the role is referenced in any policy, the operation fails and a PolicyStoreOperationNotAllowedException is thrown.

The getAppRole() method can be used to retrieve an AppRoleEntry by passing to it the Name. You can retrieve many AppRoleEntry objects by calling the getAppRoles() method and passing search criteria to it using the AppRoleSearchQuery class. Additionally, you can modify an AppRoleEntry with the modifyAppRole() method, retrieve members granted directly to an Application Role with the getDirectAppRoleMembers() method, and retrieve Application Role hierarchies for a principal with the getDirectGrantedAppRoles() method.

Revocation of the AppRoleEntry can be done using the revokeAppRole() method. Granting of the AppRoleEntry to one or more PrincipalEntry objects can be achieved statically using the grantAppRole() method or dynamically using a Role Mapping Policy.

---

**Note:** A Role Mapping Policy may define a principal (User, Group), a target (resource, resource name expression), and an (optional) Condition. Authorization Policies are used to map Application Roles to access rights. An Authorization Policy may define a principal (User, Group, Application Role), a target (resource, entitlement set, resource name expression), a Condition, and an Obligation. See Section 3.2.2.4, "Managing Role Mapping Policy (RolePolicyEntry) Objects" for more information.

Application Roles also use inheritance and hierarchy. Roles can be created in a hierarchy such that a Principal assigned to a role (using a Role Mapping Policy) also inherits any child roles (as long as it is not prohibited by other configured policies). Users who are granted actions based on a child role inherit the actions from that role's parents. Users denied actions based on a parent role are also denied actions for that role's children.

**3.2.2.4 Managing Role Mapping Policy (RolePolicyEntry) Objects**

Section 2.4.2, "Creating Role Mapping Policies" documents how to create a RolePolicyEntry object. You can also delete, modify and retrieve RolePolicyEntry...
Using Scope Levels to Manage Policy Objects

Managing Policy Objects Programmatically

objects by getting an instance of the RolePolicyManager (using getRolePolicyManager() in the ApplicationPolicy interface) and calling the appropriate method. Example 3–8 illustrates how to remove a RolePolicyEntry named TellerRoleMapping within the TellerApp ApplicationPolicy object.

Example 3–8  Using the deleteRolePolicy() Method

```java
//get the RolePolicyManager
RolePolicyManager rolePolicyManager = tellerApp.getRolePolicyManager();

//delete the RolePolicyEntry
rolePolicyManager.deleteRolePolicy("TellerRoleMapping");
```

Example 3–9 illustrates how to revise a RolePolicyEntry by passing a revised instance of the object to the modifyRolePolicy() method.

Example 3–9 Using the modifyRolePolicy() Method

```java
//get the RolePolicyManager
RolePolicyManager rolePolicyManager = tellerApp.getRolePolicyManager();

// get the policy
RolePolicyEntry rolePolicy = rolePolicyManager.getRolePolicy("TellerRoleMapping");

// change description
rolePolicy.setDescription("the policy is changed!");

//persist the change
rolePolicyManager.modifyRolePolicy(rolePolicy);
```

The getRolePolicy() method can be used to retrieve a RolePolicyEntry by passing to it the Name. You can retrieve many RolePolicyEntry objects by calling the getRolePolicies() method and passing to it an array of search criteria using the RolePolicySearchQuery class.

3.2.2.5 Managing AttributeEntry and FunctionEntry Objects

Section 2.4.3, "Creating Attribute and Function Definitions" documents how to create an AttributeEntry definition and a FunctionEntry definition for (optional) use in policy Conditions and Obligations. You can also delete, modify and retrieve these objects by calling the ExtensionManager. The following sections contain more information.

- Section 3.2.2.5.1, "Managing AttributeEntry Objects"
- Section 3.2.2.5.2, "Managing FunctionEntry Objects"

3.2.2.5.1 Managing AttributeEntry Objects

Example 3–10 retrieves an AttributeEntry object named Phone from the policy store.

```java
//get the ExtensionManager
ExtensionManager extMgr = bankApplication.getExtensionManager();

//retrieve the attribute
```
AttributeEntry<? extends DataType> oneAttrEntry = extMgr.getAttribute("Phone");

You can also retrieve many AttributeEntry objects by calling the getAttributes() method and passing search criteria to it using the AttributeSearchQuery class. Example 3–11 deletes the AttributeEntry object from the ApplicationPolicy.

**Example 3–11 Using the deleteAttribute() Method**

```
//get the ExtensionManager
ExtensionManager extMgr = bankApplication.getExtensionManager();

//retrieve the attribute
AttributeEntry<? extends DataType> oneAttrEntry = extMgr.deleteAttribute("myattr", false);
```

---

**Caution:** Remove the applicable AttributeEntry from any policies in which it is referenced before running the deleteAttribute() method. If the attribute is in use, it will not be deleted and a PolicyStoreOperationNotAllowedException will be thrown. For this release, the cascadeDelete parameter must be false.

---

To modify an AttributeEntry object, pass to the ExtensionManager the object with new, modified values using the modifyAttribute() method. Use the methods available in the AttributeEntry interface to set the new, modified values before passing the object.

### 3.2.2.5.2 Managing FunctionEntry Objects

Example 3–12 retrieves a FunctionEntry object named ClientType from the policy store. bankApplication refers to the ApplicationPolicy object from which the ExtensionManager is instantiated. ClientType refers to the unique identifier defined as the Name when the FunctionEntry object was initially created.

**Example 3–12 Using the getFunction() Method**

```
//get the ExtensionManager
ExtensionManager extMgr = bankApplication.getExtensionManager();

//retrieve the function
FunctionEntry oneFuncEntry = extMgr.getFunction("ClientType");
```

You can also retrieve many FunctionEntry objects by calling the getFunctions() method and passing search criteria to it using the FunctionSearchQuery class. Example 3–13 deletes the FunctionEntry object from the ApplicationPolicy.

**Example 3–13 Using the deleteFunction() Method**

```
//get the ExtensionManager
ExtensionManager extMgr = bankApplication.getExtensionManager();

//remove the function
extMgr.deleteFunction("ClientType", false);
```

To modify a FunctionEntry object, pass to the ExtensionManager the object with new, modified values using the modifyFunction() method. Use the methods available in the FunctionEntry interface to set the new, modified values before passing the object.
3.2.2.6 Managing ResourceEntry Objects

Section 2.3.4, "Instantiating a Resource" documents how to instantiate a ResourceEntry object from a ResourceTypeEntry object. You can also delete, modify and retrieve ResourceEntry objects by getting an instance of the ResourceManager (using getResourceManager() in the ApplicationPolicy interface, or in the PolicyDomainEntry interface if using Policy Domains to delegate administration) and calling the appropriate method.

Note: The ResourceEntry object is represented in the Oracle Entitlements Server Administration Console as a Resource.

Example 3–14 illustrates how to retrieve a ResourceEntry object. The getResource() method is defined in the ResourceFinder interface which is extended by the ResourceManager interface. By passing to the method the defined name of a resource type and the resource, a ResourceEntry will be returned.

Example 3–14 Using the getResource() Method

```java
//get the ResourceManager
ResourceManager resMgr = domain.getResourceManager();

//retrieve the Resource
ResourceEntry checkingRes = resMgr.getResource("WidgetType", "WidgetResource")
```

Example 3–15 removes a checking account ResourceEntry. domain refers to the PolicyDomainEntry object from which the ResourceManager is being retrieved. By passing to the method the defined name of a resource type and the resource, a ResourceEntry will be returned.

Example 3–15 Using deleteResource() Method

```java
//get the ResourceManager
ResourceManager resMgr = domain.getResourceManager();

//remove the Resource
resMgr.deleteResource("WidgetType", "WidgetResource", true);
```

The values of the deleteResource() parameters are defined as:

- Resource Type Name - WidgetType is the unique identifier defined as the Name when the ResourceTypeEntry was initially created.
- Name - WidgetResource is the unique identifier defined as the Name when the ResourceEntry was initially created.
- cascadeDelete - This parameter takes a value of true or false and governs how the ResourceEntry and related objects would be removed. If true, the ResourceEntry is removed from any policies that reference it. If it is the only object being referenced by a policy, the policy is also deleted. If false, and ResourceEntry instances exist, the operation fails and PolicyStoreOperationNotAllowedException is thrown.

You can also modify a ResourceEntry object by calling the modifyResource() method and passing to it a handle to the object itself in the form of an EntryReference and an array of modifications. Example 3–16 illustrates this.
Example 3–16 Using modifyResource() Method

// get the ResourceManager
ResourceManager resMgr = domain.getResourceManager();

// get resource object
ResourceEntry resEntry = resMgr.get("WidgetType", "WidgetResource");

// create attrName Attribute with value of 'test'
AttributeEntry attrEntry1 = new BasicAttributeEntry("testAttr",
    new OpssString("test");
resEntry.addResourceAttribute(attrEntry1);

// persist the change
resMgr.modifyResource(resEntry);

3.2.2.7 Managing Permission Sets

Section 2.4.4, "Defining Permission Sets" documents how to organize one or more ResourceActionsEntry objects in a PermissionSetEntry object by calling the PermissionSetManager and using the createPermissionSet() method. You can also delete, modify and retrieve PermissionSetEntry objects by getting an instance of the PermissionSetManager (using getPermissionSetManager() in the ApplicationPolicy interface, or in the PolicyDomainEntry interface if using Policy Domains to delegate administration) and calling the appropriate method.

Note: The PermissionSetEntry object is represented in the Oracle Entitlements Server Administration Console as an Entitlement.

Example 3–17 illustrates how to modify a PermissionSetEntry by removing two ResourceActionsEntry objects. domain refers to the Policy Domain under which the policy was created, and from which the PermissionSetManager is retrieved.

Example 3–17 Modifying a PermissionSetEntry

// get the PermissionSetManager
PermissionSetManager psMgr = domain.getPermissionSetManager();

// get the PermissionSet
PermissionSetEntry permSetEntry = psMgr.getPermissionSet("myPermSet");

// get the ResourceActionEntries from PermissionSet
List<ResourceActionsEntry> resultResActions = permSetEntry.getResourceActionsList();

// delete the first ResourceActionsEntry object
permSetEntry.deleteResourceActions(resultResActions.get(0));

// persist the change
psMgr.modifyPermissionSet(permSetEntry);

Example 3–18 illustrates how to remove a PermissionSetEntry object.

Example 3–18 Using the deletePermissionSet() Method

// get the PermissionSetManager
PermissionSetManager psMgr = domain.getPermissionSetManager();

// remove PermissionSetEntry
psMgr.deletePermissionSet("RptsPermSet", "true");

The values of the deletePermissionSet() parameters are defined as:

- **Name** - RptsPermSet is the unique identifier defined as the Name when the object was initially created.
- **cascadeDelete** - This parameter takes a value of true or false and governs how the PermissionSetEntry and related objects would be removed. If true, the PermissionSetEntry is removed from any policies that reference it. If it is the only object being referenced by a policy, the policy is also deleted. If false, and PermissionSetEntry instances are referenced, the operation fails and PolicyStoreOperationNotAllowedException is thrown.

The getPermissionSet() method can be used to retrieve a PermissionSetEntry, also by Name. You can retrieve many PermissionSetEntry objects by calling the getPermissionSets() method and passing search criteria to it using the PermissionSetSearchQuery class. modifyPermissionSet() will persist any changes defined in the PermissionSet object used as input.

### 3.2.2.8 Managing the Policy

Section 2.3.8, "Defining the Policy" documents how to create a PolicyEntry object by consolidating all the pieces needed to create the access control - including, but not limited to, a PolicyRuleEntry, a ResourceActionsEntry, and a PrincipalEntry; after obtaining an instance of the PolicyManager, use the createPolicy() method. You can also delete, modify and retrieve PolicyEntry objects by getting an instance of the PolicyManager (using getPolicyManager() in the ApplicationPolicy interface, or in the PolicyDomainEntry interface if using Policy Domains to delegate administration) and calling the appropriate method.

Example 3–19 illustrates how to modify the values of the Display Name and Description parameters of the PolicyEntry. domain refers to the Policy Domain under which the policy was created, and from which the PolicyManager is retrieved.

#### Example 3–19  Using modifyPolicy() Method

```java
// get the Policy
PolicyManager policyMgr = domain.getPolicyManager();
PolicyEntry policyEntry = policyMgr.getPolicy("mypolicy");

// update PolicyEntry description and displayName
policyEntry.setDescription("updated description");
policyEntry.setDisplayName("updated display name");

// persist the change
policyMgr.modifyPolicy(policyEntry);
```

Example 3–20 illustrates how to use the deletePolicy() method. Bank Policy refers to the unique identifier defined as the value of the Name parameter when the PolicyEntry was created.

#### Example 3–20  Using deletePolicy() Method

```java
PolicyManager policyMgr = domain.getPolicyManager();
policyManager.deletePolicy("BankPolicy");
```

The getPolicy() method can be used to retrieve a PolicyEntry, also by the value of its Name parameter. You can retrieve many PolicyEntry objects by calling the getPolicies() method and passing search criteria to it using the PolicySearchQuery
class. modifyPolicy() will persist any changes defined in the PolicyEntry object used as input.

To search for PolicyEntry objects, use the PolicySearchQuery class. You can build a query to search based on the following:

- Name
- Display Name
- Description
- Principal
- Permission Set
- Obligation
- Attribute
- Function

For more information, see the Oracle Fusion Middleware Management Java API Reference for Oracle Entitlements Server.

3.2.3 Managing Objects within the PolicyDomainEntry Scope

Components of policy definitions can be organized within one or more PolicyDomainEntry objects if partitioning of policies is required. These components include Resources, Permission Sets and Policies.

Note: The creation of a PolicyDomainEntry is optional. If partitioning of policies is not required, manage policy definition components at the ApplicationPolicy scope.

The following sections document how components can be managed in the ApplicationPolicy scope. These same components can be managed at the PolicyDomainEntry scope if a PolicyDomainEntry has been created for further partitioning.

- Section 3.2.2.6, "Managing ResourceEntry Objects"
- Section 3.2.2.7, "Managing Permission Sets"
- Section 3.2.2.8, "Managing the Policy"

For information on using the PolicyDomainEntry, see Section 5.7, "Delegating with a Policy Domain."
Policy distribution comprises the process used to make configured policies and policy data available to the Policy Decision Point (PDP) such that it can evaluate them and produce a GRANT or DENY authorization decision. This chapter contains the following sections.

- Section 4.1, "Understanding Policy Distribution"
- Section 4.2, "Defining Distribution Modes"
- Section 4.3, "Creating Security Module Configurations and Bindings"
- Section 4.4, "Initiating Policy Distribution"

### 4.1 Understanding Policy Distribution

Managing policies and distributing them are distinct operations in Oracle Entitlements Server. Policy management operations are used to define, modify and delete policies in the policy store. The Policy Distribution Component then makes the policies available to a PDP endpoint (Security Module) where the data is used to grant or deny access to a protected resource. Policies are not enforced until they are distributed. Policy distribution may include any or all of the following actions:

- Reading policies from the policy store.
- Caching policy objects in the in-memory policy cache maintained by the Security Module for use during authorization request processing.
- Preserving policy objects in a file-based persistent cache, local to the Policy Distribution Component, that provides independence from the policy store.

Both the central Oracle Entitlements Server Administration Console and the locally-installed (to the protected application) Security Module contain the Policy Distribution Component. This architecture allows two deployment scenarios: the first involves a centralized Policy Distribution Component that can communicate with many Security Modules while the second involves a Policy Distribution Component that is local to, and communicates with, one Security Module. The following sections contain more information.

- Section 4.1.1, "Using a Centralized Policy Distribution Component"
- Section 4.1.2, "Using a Local Policy Distribution Component"

### 4.1.1 Using a Centralized Policy Distribution Component

The centralized Policy Distribution Component scenario involves the use of the Policy Distribution Component (within the Administration Console) to act as a server
communicating with the Security Module's Policy Distribution Component client. Figure 4–1 illustrates how, in this scenario, the Security Module's Policy Distribution Component client does not communicate with the policy store. The distribution of policies is initiated by the Oracle Entitlements Server administrator and pushed to the Policy Distribution Component client. Currently, data can only be pushed in a controlled manner as described in Section 4.2.1, "Controlled Distribution." This scenario allows for a central Policy Distribution Component that can communicate with many Security Modules.

Figure 4–1 Using Oracle Entitlements Server Policy Distribution Component

4.1.2 Using a Local Policy Distribution Component

The local (to the Security Module) scenario involves the Security Module's Policy Distribution Component communicating directly with the policy store. This scenario allows for a local Policy Distribution Component to communicate with one Security Module only. The application administers management operations and decides when the Security Module instance of the Policy Distribution Component will distribute policies or policy deltas. In this deployment, as illustrated in Figure 4–2, the Policy Distribution Component pulls data from the policy store (by periodically checking the policy store for data to be distributed) and sends policy data from the policy store, making it available to the PDP after administrator-initiated policy distribution.

Figure 4–2 Using the Security Module Policy Distribution Component

Currently, data can be pulled in either a controlled manner as described in Section 4.2.1, "Controlled Distribution" or a non-controlled manner as described in Section 4.2.2, "Non-Controlled Distribution."
4.2 Defining Distribution Modes

Oracle Entitlements Server handles the task of distributing policies to individual Security Modules that protect applications and services. Policy data is distributed in either a controlled manner or a non-controlled manner. The distribution mode is defined in the jps-config.xml configuration file for each Security Module. The specified distribution mode is applicable for all ApplicationPolicy objects bound to that Security Module. The following sections have more information on the distribution modes.

- Section 4.2.1, "Controlled Distribution"
- Section 4.2.2, "Non-Controlled Distribution"

4.2.1 Controlled Distribution

Controlled distribution is initiated by the Policy Distribution Component, ensuring that the PDP client (Security Module) receives policy data that has been created or modified since the last distribution. In this respect, distribution is controlled by the policy administrator who takes explicit action to distribute the new or updated policy data. (The Policy Distribution Component maintains a versioning mechanism to keep track of policy changes and distribution.) When controlled distribution is enabled, the Security Module cannot request distribution of the Policy Distribution Component directly.

---

**Note:** The exception is when a Security Module starts and registers itself with the Policy Distribution Component with a Configuration ID. The policies are distributed to the Security Module based on this registration.

---

Controlled distribution may be push or pull. With controlled push distribution, the Policy Distribution Component distributes new and updated policy data to the Security Module where the data is stored in a local persistent cache, a file-based cache maintained by the PDP to store policy objects and provide independence from the policy store. With controlled pull distribution, a Security Module (with local Policy Distribution Component) retrieves new and updated policy data from the policy store. Controlled push distribution is the default mode.

---

**Note:** It is recommended that you use controlled-pull mode in a production environment for better performance and scalability when large policy sets are distributed to the Security Modules.

---

The Policy Distribution Component does not maintain constant live connections to its Security Module clients; it will establish a connection before distributing policy to it. Thus, the Security Module is not dependent on the policy store for making policy decisions; it can use its own local cache if the policy store is offline. When the Security Module starts, it will check if the policy store is available. If it is not available, the Security Module will use policy data from the local persistent cache.

A flush distribution of all policy data can be enforced using the flush parameter of the distributePolicy() method. Flush distribution is when the Policy Distribution Component notifies the Security Module to cleanup its locally stored policies in preparation for a new, complete re-distribution of all policy objects in the policy store. See Section 4.4, "Initiating Policy Distribution" for more information.
Creating Security Module Configurations and Bindings

With controlled distribution, if any portion of the policy distribution operation fails, the entire distribution operation fails. See Oracle Fusion Middleware Administrator’s Guide for Oracle Entitlements Server for information on the Policy Distribution Component configuration properties.

4.2.2 Non-Controlled Distribution

When the PDP client (Security Module) periodically retrieves (or pulls) policies and policy modifications from a policy store, it is referred to as non-controlled distribution. Policy changes are saved to the policy store and non-controlled distribution retrieves them at the next defined interval. Non-controlled distribution is initiated by the Security Module and may retrieve policies that are not completely defined. The policy store must be online and constantly available for non-controlled distribution. Non-controlled distribution is supported on any policy store type.

4.3 Creating Security Module Configurations and Bindings

A Security Module acts as a Policy Decision Point (PDP), receiving a request for authorization, evaluating it based on applicable policies, reaching a decision and returning the decision to the Policy Enforcement Point (PEP), the entity which first made the authorization call. In order for this process to work, the Security Module must be able to retrieve the applicable policies. This is accomplished by binding an instance of a Security Module to the appropriate ApplicationPolicy object. All Security Module instances bound to an ApplicationPolicy object will receive policy data associated with that object when policy distribution is initiated. Each Security Module instance deployed has its configuration information stored in the policy store. The SMEntry object is a pointer to the configuration information of the instance.

Note: When a Security Module starts, it registers itself with Oracle Entitlements Server. This registration record is added to the Policy Store as a PDPInfoEntry object. Registration records include the Security Module endpoint and the unique identifier that names it. The PDPInfoEntry interface is located in the oracle.security.jps.service.policystore.info.distribution package. This package also contains interfaces used to get information regarding distribution status (DistributionStatusEntry) and regarding distribution status to a particular Security Module (PDPStatusEntry).

To bind a Security Module with an ApplicationPolicy object, create an SMEntry object (representing the Security Module configuration) and bind it to the ApplicationPolicy object. Example 4–1 illustrates how to create an SMEntry object by retrieving an instance of the PolicyStore and getting the ConfigurationManager. This returns the SMEntry object which can be used for binding one or more ApplicationPolicy objects.

Example 4–1 Using the createSecurityModule() Method

//get the policy store and configuration manager

Caution: Controlled distribution is supported only on database type policy stores - not on LDAP-based policy stores. If the distribution API is invoked for an LDAP policy store, it will be non-operable.
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ConfigurationManager configMgr = ps.getConfigurationManager();

//create the SM configuration
SMEntry sm = configMgr.createSecurityModule("MyDomainSM",
    "MyDomainSM Configuration", "MyDomain Security Module Configuration");

The values of the createSecurityModule() parameters are defined as:

- **smName** - MyDomainSM is a unique identifier for the SMEntry object. The Security Module uses this value to connect to the policy store to get the configuration information. The SMEntry object itself does not contain the configuration information; it only points to it.

- **DisplayName** - MyDomainSM Configuration is an optional, human-readable name for the SMEntry object.

- **Description** - MyDomain Security Module Configuration is optional information describing the SMEntry object.

After creating it, bind the SMEntry object to a specific ApplicationPolicy object by calling the ConfigurationBindingManager interface and using the bindSecurityModule() method. Example 4–2 illustrates this step.

**Example 4–2 Using the bindSecurityModule() Method**

```java
//get the policy store and the configuration binding manager
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ConfigurationBindingManager configBindingMgr = ps.getConfigurationBindingManager();

//bind Security Module to Application Policy
configBindingMgr.bindSecurityModule("MyDomainSM", "MyAppPolicy");
```

The values of the bindSecurityModule() parameters are defined as:

- **smName** - MyDomainSM is the unique identifier defined for the SMEntry object when it was created.

- **AppID** - MyAppPolicy is the unique identifier defined for the ApplicationPolicy object when it was created.

The following sections contain information on the management methods for the Security Module configurations and bindings.

- **Section 4.3.1, "Managing Security Module Configurations"**
- **Section 4.3.2, "Managing Security Module Bindings"**

### 4.3.1 Managing Security Module Configurations

After getting an instance of the ConfigurationManager, you can also delete, retrieve and modify SMEntry objects. Example 4–3 illustrates how to get a specific Security Module configuration by passing the unique identifier of the SMEntry object.

**Example 4–3 Using the getSecurityModule() Method**

```java
//get the policy store and configuration manager
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ConfigurationManager configMgr = ps.getConfigurationManager();

//get Security Module configuration
```
SMEntry sm = configMgr.getSecurityModule("MyDomainSM");

MyDomainSM is the unique identifier defined for the SMEntry object when it was created. Additionally, you can retrieve multiple SMEntry objects by calling the getSecurityModules() method and passing to it an array of search criteria using the SecurityModuleSearchQuery class. Example 4–4 illustrates how to remove a Security Module configuration.

**Example 4–4  Using the deleteSecurityModule() Method**

```java
//get the policy store and configuration manager
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ConfigurationManager configMgr = ps.getConfigurationManager();

//get Security Module configuration
configMgr.deleteSecurityModule("MyDomainSM");
```

Again, MyDomainSM is the unique identifier defined for the SMEntry object when it was created.

### 4.3.2 Managing Security Module Bindings

After getting an instance of the ConfigurationBindingManager, you can also retrieve the ApplicationPolicy objects bound to a particular Security Module, or the Security Module bound to a particular ApplicationPolicy. Example 4–5 illustrates how to use the getBoundSecurityModules() method to retrieve the identifier for all SMEntry objects bound to a particular ApplicationPolicy object.

**Example 4–5  Using the getBoundSecurityModules() Method**

```java
//get the policy store and the configuration binding manager
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ConfigurationBindingManager configBindingMgr = ps.getConfigurationBindingManager();

//get Security Module bound to Application Policy
List<SMEntry> sms = configBindingMgr.getBoundSecurityModules("MyAppPolicy");
```

MyAppPolicy is the unique identifier defined for the ApplicationPolicy object when it was created. The getBoundSecurityModules() method returns a list of the unique identifiers for all SMEntry objects bound to the ApplicationPolicy. Example 4–6 illustrates the reverse: retrieving all ApplicationPolicy objects bound to a particular Security Module.

**Example 4–6  Using the getBoundApplications() Method**

```java
//get the policy store and the configuration binding manager
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ConfigurationBindingManager configBindingMgr = ps.getConfigurationBindingManager();

//get Application Policy bound to Security Module
List<ApplicationPolicy> apps = configBindingMgr.getBoundApplications("MyDomainSM");
```

MyDomainSM is the unique identifier defined for the SMEntry object when it was created. The getBoundApplications() method returns a list of the unique identifiers for all ApplicationPolicy objects bound to the SMEntry. Example 4–7 illustrates how to unbind an SMEntry object from its partner ApplicationPolicy object.
**Example 4–7 Using the unbindSM() Method**

```java
//get the policy store and the configuration binding manager
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ConfigurationBindingManager configBindingMgr = ps.getConfigurationBindingManager();

//unbind Application Policy from Security Module
configBindingMgr.unbindSM("MyDomainSM", "MyAppPolicy");
```

MyDomainSM is the unique identifier defined for the SMEntry object when it was created. MyAppPolicy is the unique identifier defined for the ApplicationPolicy object when it was created.

### 4.4 Initiating Policy Distribution

Programmatically, policy distribution is performed by calling the `distributePolicy()` method. This method distributes the policies created for an ApplicationPolicy object to the Security Module that is bound to it. A PDP endpoint receives only those policies which are bound to it. **Example 4–8** illustrates how to call the PolicyDistributionManager and use the `distributePolicy()` method. It also includes code to check the status of the distribution and to wait until the operation is 100% complete.

**Example 4–8 Using the distributePolicy() Method**

```java
//get the application policy
PolicyStore ps = ctx.getServiceInstance(PolicyStore.class);
ApplicationPolicy bankApplication = ps.getApplicationPolicy("AcmeBank");

//get the PolicyDistributionManager
PolicyDistributionManager pdm = bankApplication.getPolicyDistributionManager();

//distribute policies
String distID = pdm.distributePolicy(true);
DistributionStatusEntry status = pdm.getDistributionStatus(distID);

while (status.getPercentComplete() != 100) {
    Thread.currentThread().sleep(200);
    status = pdm.getDistributionStatus(distID);
}
```

Note the *flush* parameter of `distributePolicy()` is set to true. This indicates that the policies will be distributed in a *flush* manner. In other words, the Policy Distribution Component informs the Security Module to cleanup its locally stored policies in preparation for a new, complete re-distribution of all policy objects in the policy store. A value of false indicates an incremental distribution of policies when only deltas are distributed.

The `distributePolicy()` method returns a distribution identifier string that can be passed to the application using the `getDistributionStatus()` method to query the progress of the distribution.
Initiating Policy Distribution

Note: `distributePolicy()` is an asynchronous method; if the application is stopped before the distribution is complete, the distribution process will be interrupted.

A second `getDistributionStatus()` method takes as input a start time and an end time. It returns a list of `DistributionStatusEntry` objects. A `DistributionStatusEntry` object represents the distribution status (complete or in progress) and includes a start time, an end time, the distribution initiator, and whether the distribution is successful or not for each PDP.
System administrative rights and policy management permissions can be delegated from one administrator to another by creating Administration Roles with restricted rights, or by granting an existing Administration Role to a user. Delegating administration consists of defining a role with a subject (the person to whom the role is granted), resources (the objects to which the role pertains) and the actions on the resource that the role controls (view, manage). This chapter documents information on how to delegate policy and system administrative tasks. It contains the following sections.

- Section 5.1, "Delegating Administration"
- Section 5.2, "Managing Scope and Delegating Granularity"
- Section 5.3, "Assigning Permissions"
- Section 5.4, "Using the Default Administration Roles"
- Section 5.5, "Creating Administration Roles"
- Section 5.6, "Managing Administration Roles"
- Section 5.7, "Delegating with a Policy Domain"

### 5.1 Delegating Administration

*Administration* is when one or more authorized rights are granted to someone to do a certain job. *Delegation* is the ability for that someone to transfer the authorized right that has been granted them to another. In combination, we can define *delegating administration* as the transference of authorized rights from one to another. In Oracle Entitlements Server, administrators who are authorized to perform a task on policy objects and entities may transfer this right to others.

Delegated administration in Oracle Entitlements Server is modelled using the Role-Based Access Control (RBAC) approach. This approach allows users to transfer the administration of Applications, Policy Domains, and other policy objects using roles. The core concept behind RBAC is that privileges (approvals to perform an action) are coupled with the objects on which the action can be performed and modelled as *permissions*. These permissions are then assigned to roles. When users are assigned the roles, the user is granted the appropriate permissions.
Managing Scope and Delegating Granularity

Figure 5–1  The Administration Role Model

As illustrated in Figure 5–1, an Administration Role is created for a particular operation on a policy related object. The permissions to perform the operation specific to that job are defined in that role. Users are then assigned the role and through those assignments acquire the permissions to perform the job. As users are not directly assigned permissions, management of individual user privileges is a matter of assigning the appropriate roles to the appropriate users. Administration Roles are used to determine who may manage policy objects.

5.2 Managing Scope and Delegating Granularity

Delegated administration is all about transferring management of resources and policy objects from one person to another. The **scope** of the delegation (or range of objects covered by the delegation) is defined in levels. The **granularity** of administration defines the type of objects managed at each scope. A default Administration Role is automatically created when each scope is created; additional Administration Roles can be created later. From highest to lowest, the scopes and applicable granularity are as follows:

- The top-level System Administrator has privileges to manage system-level resources as well as policy-related objects at the top-level Policy Store scope. System resources include Administrator Roles and system configurations and bindings. Objects at the Policy Store level are the ApplicationPolicy objects and global objects.

  **Note:** System Administrators have rights to the entire Policy Store, including all ApplicationPolicy objects and child PolicyDomain objects but they are primarily intended to manage configurations, ApplicationPolicy objects, and the bindings between the two.

- Application Policy administrators have privileges to manage all objects in the ApplicationPolicy object to which it is assigned. One Application Policy Administrator is generated for each Application Policy that is created. They are primarily intended to delegate the management of policy objects within the Application Policy (including the Policy Domain objects and its children, such as Functions, Attributes, Application Roles and Resource Types).

- Policy Domain administrators have privileges to manage all child objects in the Policy Domain object to which it is assigned. One Policy Domain Administrator is generated for each Policy Domain that is created. They are primarily intended to define the policies, permission sets, and resources within the applicable Policy Domain.

  **Note:** See Chapter 1, "Using the Policy Model" for more information on the ApplicationPolicy objects and Section 5.7, "Delegating with a Policy Domain" for information on the PolicyDomain objects.
5.3 Assigning Permissions

Administration Roles can be assigned permissions with Manage or View actions. The privileges of these actions are:

- Administrator Roles with Manage privileges may call all methods on objects in the assigned administrative scope including any child objects. For example, an Application Policy administrator with Manage rights may call all methods on objects in both the Application Policy and its Policy Domain objects. An administrator with Manage rights may also view any required objects in a parent scope. For example, an administrator with Manage rights in a Policy Domain can view all Resources Types, Functions, and Attributes in its parent Application Policy because these objects are used when defining policies.

- Administrators with View privileges may call only get methods in the assigned administrative scope including any child objects. For example, a Global administrator with View privileges may view all objects in all Application Policy objects and its Policy Domain objects.

5.4 Using the Default Administration Roles

After installation, the Policy Store will contain a default Administration Role called SystemAdmin with full view and manage rights at the Policy Store level. This and other default administration roles are described in the following list. Only the members of these default Administration Roles can create and manage other Administration Roles. The default Administration Roles cannot be deleted and their rights cannot be changed.

- SystemAdmin — This is the default Policy Store administrator with Manage rights in the entire Policy Store. This role is assigned to the WebLogic Server Administrators group, and has all the rights needed to manage policies in all Application Policy objects and Policy Domain objects.

- ApplicationPolicyAdmin — A role by this name is automatically created with each new Application Policy object. It has Manage rights in the Application Policy and its nested Policy Domain objects.

- PolicyDomainAdmin — A role by this name is automatically created with each new Policy Domain object. It has Manage rights in the Policy Domain and any nested Policy Domain objects. It also has View rights on objects in its parent Application Policy.

5.5 Creating Administration Roles

Administration Roles are used to delegate system administrative rights. An Administration Role can be created for purposes of managing data at different scopes. For example, Application Policy and Policy Domain administrators can be defined by creating an Administration Role at the appropriate level and assigning the role to a user or a group.

\[ \text{Note: Administration Roles delegate system privileges through scoping and are not hierarchical. See Section 5.2, "Managing Scope and Delegating Granularity" for more information.} \]
Creating Administration Roles involves a number of specifics. Use the following steps as a blueprint to grant View or Manage permissions on specific administration resources.

1. Retrieve the object within which the Administration Role will be created and an instance of the AdminManager as documented in Section 5.5.1, "Creating An Administration Role."

2. Define the resource and appropriate actions as documented in Section 5.5.2, "Assigning Actions and Resources (Permissions) to an Administration Role."

3. Assign users (principals) as documented in Section 5.5.3, "Assigning Principals to an Administration Role."

Section 5.5.4, "Retrieving a Principal's Administration Resources" contains information on how to retrieve the administration roles that a principal has been assigned.

5.5.1 Creating An Administration Role

To create an Administration Role, retrieve the object that comprises the desired management scope (Policy Store, Application Policy or Policy Domain), use the getAdminManager() method to retrieve an instance of the AdminManager, and then use the createAdminRole() method to create the adminRole role. The following code illustrates the creation of an administrator named AppAdmin for the TRADING Application Policy.

```java
//Get the Application Policy and AdminManager
ApplicationPolicy app = ps.getApplicationPolicy("TRADING");
AdminManager appAdminManager = app.getAdminManager();
AdminRoleEntry adminRole = appAdminManager.createAdminRole
("AppAdmin", "AppAdmin Role", "Role for application admins.");
```

The values of the createAdminRole() parameters are defined as follows:

- Name - AppAdmin is the name of the Administration Role.
- Display Name - AppAdmin Role is an optional, human-readable name for the Administration Role.
- Description - Role for application admins. is an optional description of the Administration Role.

5.5.2 Assigning Actions and Resources (Permissions) to an Administration Role

Privileges are assigned to an Administration Role by creating an ArrayList into which the resource(s) being managed and the permitted actions are added (using a BasicAdminResourceActionEntry). In the following code, the previously created AppAdmin role is assigned Manage rights on Resource Types and Application Roles in the TRADING application.

```java
//Construct the permission to be granted
List<AdminResourceActionEntry> adminResourceActions = new ArrayList
<AdminResourceActionEntry>();

//Add operations (Manage) and objects (resources) to the permission
adminResourceActions.add(new BasicAdminResourceActionEntry
(AdminResource.RESOURCE_TYPE, Action.MANAGE));
adminResourceActions.add(new BasicAdminResourceActionEntry
(AdminResource.APPLICATION_ROLE, Action.MANAGE));

//Grant AppAdmin the rights
```
admManager.grant(adminRole, adminResourceActions);

To remove privileges from a role, use the `revoke()` method rather than the `grant()` method. The allowed resource name options for the Policy Store, Application Policy, and Policy Domain scopes are described in Table 5–1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADMIN_POLICY</td>
<td>Allows management of Administration Policy</td>
</tr>
<tr>
<td>ADMIN_ROLE</td>
<td>Allows management of Administration Role membership and permissions</td>
</tr>
<tr>
<td>APPLICATION_POLICY</td>
<td>Allows management of Application Policy objects</td>
</tr>
<tr>
<td>APPLICATION_ROLE</td>
<td>Allows management of Application Roles</td>
</tr>
<tr>
<td>CONFIGURATION</td>
<td>Allows management of Security Modules</td>
</tr>
<tr>
<td>DISTRIBUTE_APPLICATION_POLICY</td>
<td>Allows administrator to initiate policy distribution</td>
</tr>
<tr>
<td>ENROLL</td>
<td>Allows administrator to enroll a Security Module instance</td>
</tr>
<tr>
<td>EXTENSION</td>
<td>Allows management of Functions and Attributes</td>
</tr>
<tr>
<td>PERMISSION_SET</td>
<td>Allows management of Permission Sets</td>
</tr>
<tr>
<td>POLICY</td>
<td>Allows management of Policies</td>
</tr>
<tr>
<td>RESOURCE_TYPE</td>
<td>Allows management of Resource Types</td>
</tr>
<tr>
<td>RESOURCE</td>
<td>Allows management of Resources</td>
</tr>
<tr>
<td>ROLE_CATEGORY</td>
<td>Allows management of Role Categories</td>
</tr>
<tr>
<td>SUB_POLICY_DOMAIN</td>
<td>Allows management of child Policy Domain objects</td>
</tr>
</tbody>
</table>

5.5.3 Assigning Principals to an Administration Role

One or more principals are assigned to the Administration Role by creating a second `ArrayList` with the appropriate user entries and passing the list to the `grantAdminRole()` method. In the following code, the previously created `adminRole` role is granted to the user SMITH.

```java
//Construct the list of users to be granted
List<PrincipalEntry> principals = new ArrayList<PrincipalEntry>();
principals.add(new BasicPrincipalEntry
{'weblogic.security.principal.WLSUserImpl', "SMITH"});

//Grant the users in the list the role
adminManager.grantAdminRole(adminRole, principals);
```

To remove principals from a role, use the `revokeAdminRole()` method.

5.5.4 Retrieving a Principal's Administration Resources

To determine what resources an administrative user can access, get an instance of the `AdminManager` at the appropriate scope (Policy Store, Application Policy, or Policy Domain) and use the `getAdminRole()` method and name of the Administration Role to retrieve the administrator. Then by invoking the `getGrantedAdminResources()` method, all `AdminResourceActionEntry` objects applicable to the administrator will be returned. (A `AdminResourceActionEntry` object pairs an entity that can be managed by the administrator with the action that can be performed on it.)
5.6 Managing Administration Roles

Section 5.5, "Creating Administration Roles" documented how to create an AdminRoleEntry object. Administration Roles can be created at all scope levels (including the PolicyStore, ApplicationPolicy and PolicyDomain) by retrieving an instance of the AdminManager from within the desired scope. You can also delete and retrieve AdminRoleEntry objects from any of these scopes by getting an instance of the AdminManager. Example 5–1 illustrates the delete action by getting the AdminManager in an ApplicationPolicy.

Example 5–1 Using deleteAdminRole() Method

```java
//Get the Application Policy and AdminManager
ApplicationPolicy app = ps.getApplicationPolicy("TRADING");
AdminManager appAdminManager = app.getAdminManager();

//delete the Administration Role
AdminRoleEntry adminRole = appAdminManager.deleteAdminRole("AppAdmin");

TRADING is the name of the ApplicationPolicy under which the AdminRoleEntry object was created. AppAdmin is the unique identifier of the role being deleted.

The getAdminRole() method can be used to retrieve an AdminRoleEntry, also by Name. Example 5–2 illustrates this.

Example 5–2 Using getAdminRole() Method

```java
//Get the Application Policy and AdminManager
ApplicationPolicy app = ps.getApplicationPolicy("TRADING");
AdminManager appAdminManager = app.getAdminManager();

//Get the Administration Role
AdminRoleEntry adminRole = appAdminManager.getAdminRole("AppAdmin");

You can retrieve many AdminRoleEntry objects by calling the getAdminRoles() method and passing search criteria to it using the ResourceTypeSearchQuery class. Also available in the AdminManager interface are methods that do the following:

- Add or remove a PrincipalEntry object as an administration role member.
- Return a list of PrincipalEntry objects granted the named administration role.
- Grant or revoke actions and resources (AdminResourceActionEntry) for the named administration role.
- Retrieve the actions and resources (AdminResourceActionEntry) defined for the current administrator.
- Modify the administration role.

5.7 Delegating with a Policy Domain

A Policy Domain contains the components of completed policy definitions. It is the amalgamation of a target Resource (an instance of the Resource Type), a Permission Set (the actions that can be performed on the Resource), and a Policy (a rule that assembles the controls and the principals they affect). Policy Domains are created for purposes of delegating administration. One (or more) of these domains can be created to delegate policy management to different administrators.
Delegating with a Policy Domain

Delegating Policy Administration

Administration of the policies securing one protected application may be delegated using one or more Policy Domains. The use of multiple Policy Domains allows policies to be partitioned according to defined logic, such as the architecture of the protected application or how administration of the policies will be delegated. For example, one Policy Domain can be used to maintain all policies securing a Resource or multiple Policy Domains can be used to reflect a particular characteristic of the Resource. Different administrators can then be placed in charge of different Policy Domains. If there is no need to delegate policy administration, there is no need to create any Policy Domains. In this case, all child objects associated with a Policy Domain can be created by calling the applicable child object manager using the ApplicationPolicy interface.

The Policy Domain is programmatically represented as a PolicyDomainEntry object. Within an ApplicationPolicy object, one or more (optional) PolicyDomainEntry objects can be created. A PolicyDomainEntry object may contain one or more child objects. These objects need to be defined before creating the Policy Domain.

To create a PolicyDomainEntry, obtain an instance of the PolicyDomainManager using getPolicyDomainManager(). (You can invoke getPolicyDomainManager() for an ApplicationPolicy or for a PolicyDomainEntry itself to create nested Policy Domains.) Use the createPolicyDomain() method of the PolicyDomainManager interface to create the object. Example 5–3 creates a PolicyDomainEntry object named East_Trading by retrieving the PolicyDomainManager from the Trading ApplicationPolicy.

Example 5–3 Using createPolicyDomain() Method

```
PolicyDomainManager domainMgr = Trading.getPolicyDomainManager();
PolicyDomainEntry domain = domainMgr.createPolicyDomain
    ("East_Trading", "East_Trading Domain", "East_Trading Domain");
```

The values of the createPolicyDomain() parameters are defined as:

- **Name** - East_Trading is a unique identifier for the PolicyDomainEntry.
- **Display Name** - East_Trading Domain is an optional, human-readable name for the PolicyDomainEntry object.
- **Description** - East_Trading Domain is optional information describing the PolicyDomainEntry object.

After creating a PolicyDomainEntry object, the necessary child objects can be added to it thus allowing the administrator the control in creating policy definition components. The following list documents the child objects of a PolicyDomainEntry with pointers to the appropriate descriptive section in Chapter 2, "Constructing A Policy Programmatically."

---

**Note:** Because the creation of a Policy Domain is optional, an ApplicationPolicy object can serve as a default Policy Domain under which a Resource, a Permission Set, and a Policy can be created. Creation of subsequent Policy Domains is dependent on the organization’s plan for delegation.

**Caution:** Deleting a PolicyDomainEntry object deletes all child objects created within it.
Delegating with a Policy Domain

- A PermissionSetEntry (one or more ResourceActionsEntry objects that associate a specific resource with the actions that can be performed on it). See Section 2.4.4, "Defining Permission Sets" for more information.

- A PolicyEntry (includes one PolicyRuleEntry, one PermissionSetEntry, one PrincipalEntry or AppRoleEntry and, optionally, one ObligationEntry). See Section 2.3.8, "Defining the Policy" for more information.

- An AdminRoleEntry (to define management of the domain). See Section 2.3.4, "Instantiating a Resource" for more information.

---

**Note:** The same target Resource cannot be shared between Policy Domains.
Handling Authorization Calls and Decisions

Oracle Entitlements Server contains different application programming interfaces (API) that allow the caller to request authorization for a particular subject and handle the returned decisions. This chapter contains the following sections.

- Using the PEP API
- Making checkPermission() Calls
- Using the XACML Gateway

6.1 Using the PEP API

The AzAPI is a Java API developed by the OpenAZ project and designed to communicate requests for authorization decisions and responses to same. The communications are based on the authorization decision request and response standards defined in the XACML specifications and require that an authorization engine create request and response objects using these definitions. The AzAPI interfaces enable a Policy Decision Point (PDP) to supply and consume all the XACML information required when submitting an authorization request and receiving an authorization response.

Note: More information on the OpenAZ project can be found at http://openliberty.org/.

The Oracle Entitlements Server PEP API are built on top of the AzAPI for protecting either Java or .NET applications. The PEP API contain utility classes for building a Policy Enforcement Point (PEP), and are designed to present a more simplified, scalable interface than the AzAPI, using native Java or .NET data objects rather than XACML data objects. Figure 6–1 illustrates the relationship between the AzAPI, the PEP API and Oracle Entitlements Server.
The following sections contain information on how the AzAPI has been implemented by Oracle Entitlements Server.

- Section 6.1.1, "Understanding the PEP API"
- Section 6.1.2, "Using the PEP API for Java"
- Section 6.1.3, "Using the PEP API for .NET"

### 6.1.1 Understanding the PEP API

Oracle Entitlements Server provides a Java and a .NET provider implementation of the `org.openliberty.openaz.azapi.pep` package. The Java PEP API provider is packaged in `oracle.security.jps.openaz.pep`. All .NET provider interfaces are implemented under the `Oes.Pep.Api` name space.

| Note: | The PEP API call the Web Services Security Module through its proxy implementation. This proxy layer will handle failover and caching. For information on installing the Oracle Entitlements Server Web Service Security Module, see Oracle Fusion Middleware Installation Guide for Oracle Identity and Access Management. |

The following sections contain more information.

- Section 6.1.1.1, "Working with the PEP API"
- Section 6.1.1.2, "Using the Different PEP API Requests"
- Section 6.1.1.3, "Specifying the PEP API Subject"
- Section 6.1.1.4, "Formatting Authorization Request Strings"
- Section 6.1.1.5, "Retrieving Authentication Information Using a Query"

### 6.1.1.1 Working with the PEP API

For each PEP API authorization request, the provider implementation will invoke the Oracle Entitlements Server Authorization Engine and return a decision. The implementation is responsible for converting and mapping native Java or .NET objects (subjects, resources, actions and the like) to the underlying security platform. The provider implementation takes the following actions to retrieve an authorization decision. This sequence assumes policies have already been defined for the specified target. The Request and the Response provide a standard format for interacting with a PDP.
1. A request for access is received at the PEP.

   A request contains a Subject, Resource, Action, and (optional) Environment attributes. It can be a call to authorize a Subject to access a protected resource, or it may query for what is allowed in terms of actions and authorization decisions.

2. The PEP calls the PEP API which passes a request for authorization (based on the type of Resource being accessed) to the Policy Decision Point (PDP).

   The PEP API determines the type of authorization request based on the Resource Type.

3. The PDP requests additional subject, resource, action, environment and other attributes, if applicable.

   Attributes are requested from a Policy Information Point (PIP). The PIP returns the requested attributes and the information is passed back to the PDP.

4. The PDP evaluates the request against any relevant policies.

5. The PDP returns a response (including the authorization decision) to the PEP.

   A response consists of one or more results. (Multiple results can only be caused by evaluation of a hierarchical resource.) Each result contains a Decision (Permit, Deny), status information (for example, why the evaluation failed) and (optionally) one or more Obligations (the PEP is obligated to act on these before granting or denying access).

6. The PEP fulfills the obligations, if any.

7. The PEP permits access to the resource if the request is granted; otherwise, access is denied.

6.1.1.2 Using the Different PEP API Requests

The PEP API can be used to request access authorization in several ways. One or more subjects can be authorized in a single or bulk authorization call, respectively. Oracle Entitlements Server also supports queries on authorization decisions and actions. Table 6–1, "Understanding the PEP API Authorization Request Types" contains more details on these request types.

<table>
<thead>
<tr>
<th>Authorization Request Type</th>
<th>Input Parameters</th>
<th>Returns</th>
</tr>
</thead>
</table>
| Single Authorization Request | Object or string representing the Subject  
| (Only supported when using permissions) | Object representing a Resource/Action pair or a Resource and an Action  
| | Object representing the environment context (optional) |
| Bulk Authorization Request | Object or string representing one subject  
| | List representing multiple resource/action pairs, or one list representing a resource and one representing an action  
| | Object representing the environment context |

<table>
<thead>
<tr>
<th></th>
<th>Returns</th>
</tr>
</thead>
</table>
| | Returns a Response object with an authorization decision and (optional) Obligation for a single request  
| | Throws an exception |
| | Returns a Response object with multiple authorization decisions and (optional) Obligations for multiple requests in a single call  
| | Throws an exception |
### 6.1.1.3 Specifying the PEP API Subject

The Subject requesting authorization to access a protected resource can be defined by passing a string, or an object representing a currently authenticated user.

- A String representing the value of the Name of the subject as defined in the identity store may be passed; for example, Josh Smith.
- A Java or .NET object (representing an authenticated user) may be passed. See Section 6.1.2.1, "Getting a Java PEP API Subject" or Section 6.1.3.1, "Getting a .NET PEP API Subject" for details.

### 6.1.1.4 Formatting Authorization Request Strings

The PEP API methods contain a string that defines the scope of the request. A resource string is then defined within the scope string. The following sections contain information on how to format these scope and resource strings.

- **Section 6.1.1.4.1, "Formatting the PEP API Scope String"**
- **Section 6.1.1.4.2, "Formatting the PEP API Resource String"**

#### 6.1.1.4.1 Formatting the PEP API Scope String

The scope input string is a PDP policy-specific resource representation that encapsulates resource, actions and search scope information. It is represented as:

```java
String scope = "resource = resourceString, actions = actionString1, actionString2, actionString3, searchscope = immediate/children";
```
The following is true regarding this representation.

- **resource** is required and the resource string should appear first within the scope string. See Section 6.1.1.4.2, "Formatting the PEP API Resource String."
- **actions** is optional. If present, it contains a comma separated list of requested actions and they must precede scope.
- **searchscope** is optional, is only used with the Query request type, and takes a value of *children* (the default value) or *immediate*.
  - If the value is *children*, **resourceString** may contain only the application identifier as documented in Section 6.1.1.4.2, "Formatting the PEP API Resource String." In this case, the PEP API provider will query the specified resource object and its children (if any). In the following example, Scope string defines a resource which contains a Resource string (with application identifier), no actions and no defined search scope; thus, the search scope is set to *children*, by default.
    
    ```
    String scope = "resource = PepQueryTest/resource_type_1/resource_1";
    ```
  - If the value is *immediate*, **resourceString** should be fully qualified as documented in Section 6.1.1.4.2, "Formatting the PEP API Resource String." In this case, the PEP API provider will query the specified resource object. For example:
    
    ```
    String scope = "resource = PepQueryTest/resource_type_1/resource_1, actions = action1,action2, searchscope=immediate";
    ```

The following Scope string defines a hierarchical resource.

```
String scope= "resource = PepQueryTest/hierarchical_type//res1/res2/res3, searchscope= children";
```

### 6.1.1.4.2 Formatting the PEP API Resource String

The string should be in the format `appId/resourceType/resourceName` with the forward slash (/) acting as delimiter. The appId and resourceType cannot be empty or null; the resourceName can be empty for a query request but must be populated for all other requests.

When formatting the string, there is no need to escape the delimiter character if it is used in the resourceName. For example, if there is a hierarchical resource with the name `/res1/res2/res3`, the resource string passed to the PEP API will be `appId/ResType/res1/res2/res3`. It is necessary to escape the delimiter character if it is used in the appId or resourceType though. In these cases, a string with more than two delimiters is considered invalid.

---

**Note:** The Resource string `myapp/computer\laptop/res1` is invalid because the character after `\` is neither `/` nor `\`.  

---

The following list of examples illustrate these rules.

- **myapp/computer\laptop/res1** signifies a Resource named res1 in the myapp Application with the Resource Type computer/laptop.
- **myapp/computer\laptop/res1** signifies a Resource named res1 in the myapp Application with the Resource Type computer/laptop.
- `myapp/computer/laptop/res1` signifies a Resource named `laptop/res1` in the `myapp` Application with the Resource Type `computer`.

- `myapp/computer/laptop/res1` signifies a Resource named `laptop/res1` in the `myapp` Application with the Resource Type `computer`.


**Note:** For strings in Java, the character `\` itself needs to be escaped. Thus, three of the strings previously documented, in Java, are:

- `myapp/computer/laptop/mybox`
- `myapp/computer/laptop/mybox`
- `myapp/computer/laptop/mybox`

### 6.1.1.5 Retrieving Authentication Information Using a Query

A Query operation can be initiated using the PEP API. The query accepts the following objects as input parameters, returning the appropriate results.

- **Application and Resource Type** - results are returned for all Resource instances for the Resource Type in the given Application. This call is valid for both hierarchical and non-hierarchical Resource Types.

- **Application, Resource Type and Resource** - results are returned for the given Resource instances for the Resource Type in the given Application. If the Resource Type is hierarchical, results are returned for the children of the given Resource instance as well.

There are two types of Queries as described in Table 6–2. Each type of query can operate on hierarchical and non-hierarchical Resource instances.

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Returns</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns allowed/denied actions on a Resource and/or its children (depending on the defined scope)</td>
<td>Returns a PepRequest that contains privileges of the specified subject.</td>
<td>public PepRequest newQueryPepRequest(Object subjectObj, Object environmentObj, String scope, PepRequestQueryType queryType) throws PepException</td>
</tr>
<tr>
<td>Returns full authorization results on a Resource and its children.</td>
<td>Returns full authorization results on a Resource and its children. Returns a PepRequest that can be used to get resource-actions that are allowed or denied access to current subject with given environment parameter.</td>
<td>public PepRequest newQueryPepRequest(Object subjectObj, Object resourceObj, Object actionObj, Object environmentObj, PepResponseType responseType, boolean resourceActionOnly) throws PepException</td>
</tr>
</tbody>
</table>
6.2 Using the PEP API for Java

The following sections contain information on how to use the PEP API for Java.

- Section 6.1.2.1, "Getting a Java PEP API Subject"
- Section 6.1.2.2, "Making Simple Java PEP API Authorization Requests"
- Section 6.1.2.3, "Processing Java PEP API Obligations"
- Section 6.1.2.4, "Making Java PEP API Bulk Authorization Requests"
- Section 6.1.2.5, "Making Java PEP API Query Requests"
- Section 6.1.2.6, "Configuring the Java PEP API"

6.1.2.1 Getting a Java PEP API Subject

This section contains sample code that illustrates how the PEP API for Java can retrieve an authenticated subject for an authorization request. It is recommended to call the `newPepRequest()` method with a Java Authentication and Authorization Service (JAAS) Subject.

```
Note: A string subject will be converted to a JAAS subject.
```

See Section 6.1.1.4, "Formatting Authorization Request Strings" for information on how the scope and resource strings are specified in the following examples.

- Example 6–1, "Using Authenticated Subject in Java PEP API Request"
- Example 6–2, "Using WebLogic Server Subject with Java PEP API Request"
- Example 6–3, "Using Websphere Application Server Subject with Java PEP API Request"

Example 6–1 shows how to get the authenticated user with the login service and use the authenticated subject for a single PEP API authorization request. This code returns an authorization decision based on the specified resource and action. (Bulk and query requests are created and used in a similar way.)

**Example 6–1 Using Authenticated Subject in Java PEP API Request**

```java
ServiceLocator locator = JpsServiceLocator.getServiceLocator();
LoginService loginService = locator.lookup(LoginService.class);
CallbackHandler cbh = new MyCallbackHandler("name", "password".toCharArray());
LoginContext ctx = loginService.getLoginContext(new Subject(), cbh);
ctx.login();
Subject s = ctx.getSubject();
String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
Map<String, String> env = new HashMap<String, String>();
env.put("myAttr", "Hello");

//Alternately, a String subject like "tom" can be used.
String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
Map<String, String> env = new HashMap<String, String>();
env.put("myAttr", "Hello");

PepRequest pepRequest =
```

Note: A string subject will be converted to a JAAS subject.
PepRequestFactoryImpl.getPepRequestFactory().newPepRequest
(s, action, resourceString, env);

PepResponse response = pepRequest.decide();
System.out.println("result: " + response.allowed());
Map<String, Obligation> obligations = response.getObligations();
for (String name : obligations.keySet())
{
    System.out.print("obligation: name = " + name + ", values = " +
    obligations.get(name).getStringValues());
}

Example 6-2 illustrates how, after Java Enterprise Edition (JEE) authentication, you can get the WebLogic Server subject to use with the PEP API.

Example 6-2 Using WebLogic Server Subject with Java PEP API Request
import weblogic.security.Security;
...

Subject s = Security.getCurrentSubject();

String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
Map<String, String> env = new HashMap<String, String>();
env.put("myAttr", "Hello");

PepResponse response = PepRequestFactoryImpl.getPepRequestFactory().newPepRequest(s, action, resourceString, env).decide();
System.out.println("result: " + response.allowed());
Map<String, Obligation> obligations = response.getObligations();
for (String name : obligations.keySet())
{
    System.out.print("obligation: name = " + name + ", values = " +
    obligations.get(name).getStringValues());
}

Example 6-3 illustrates how, after Java Enterprise Edition (JEE) authentication, you can get the Websphere Application Server subject to use with the PEP API.

Example 6-3 Using Websphere Application Server Subject with Java PEP API Request
import com.ibm.websphere.security.auth.WSSubject;
...

Subject s = WSSubject.getCallerSubject();

String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
Map<String, String> env = new HashMap<String, String>();
env.put("myAttr", "Hello");

PepResponse response = PepRequestFactoryImpl.getPepRequestFactory().newPepRequest(s, action, resourceString, env).decide();
System.out.println("result: " + response.allowed());
Map<String, Obligation> obligations = response.getObligations();
for (String name : obligations.keySet())
{
    System.out.print("obligation: name = " + name + ", values = " +

Using the PEP API

Handling Authorization Calls and Decisions

Example 6-4 illustrates how, after Java Enterprise Edition (JEE) authentication, you can get the Websphere Application Server subject to use with the PEP API.

Example 6-4 Using OPSS Subject with Java PEP API Request

```java
import oracle.security.jps.util.SubjectUtil;

...

// Assuming the OPSS filter/interceptor were configured,
// SubjectUtil.getCurrentSubject would return the subject
// on JRF (WAS or WLS)
Subject s = SubjectUtil.getCurrentSubject();

String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
Map<String, String> env = new HashMap<String, String>();
env.put("myAttr", "Hello");

PepResponse response = PepRequestFactoryImpl.getPepRequestFactory().
    newPepRequest(s, action, resourceString, env).decide();
System.out.println("result: " + response.allowed());
Map<String, Obligation> obligations = response.getObligations();
for (String name : obligations.keySet())
{
    System.out.print("obligation: name= " + name + ", values = "
        + obligations.get(name).getStringValues());
}
```

6.1.2.2 Making Simple Java PEP API Authorization Requests

Oracle Entitlements Server offers two types of query requests. You can request a list of all actions for a particular Resource (and its children), or you can request complete authorization results for a particular Resource (and its children). Both types of queries will retrieve results for all instantiated Resources of a given Resource Type. Example 6-5 is a query request against a particular Resource. Note that the search scope is defined as *immediate*. See Section 6.1.1.4, "Formatting Authorization Request Strings" for information on how the scope and resource strings are specified.

Example 6-5 Requesting Java PEP API Authorization Against a Resource

```java
...

String scope = "resource = PepQueryTest/resource_type_1/resource_1,
    actions = action1, searchscope=immediate";
PepRequest req = PepRequestFactoryImpl.getPepRequestFactory().
    newQueryPepRequest(subject, env, scope,
    PepRequestQueryType.RETURN_ONLY_ALLOWED_RESULTS);

PepResponse resp = req.decide();

//List of RuntimeAction objects
List actions = (List) resp.getAction();
RuntimeResource resource = (RuntimeResource) resp.getResource();
```
6.1.2.3 Processing Java PEP API Obligations

An Obligation specifies optional information that is returned to the calling application with the access decision. Each obligation in the PEP API response has a map in type `Map<String, String>` (There are no double quotes around the String value.) See Section 6.1.1.4, "Formatting Authorization Request Strings" for information on how the scope and resource strings are specified in the following examples.

- Example 6–6, "Making PEP API Request and Parsing Response"
- Example 6–7, "Returned Obligations from Example 6–6"

Example 6–6 is an authorization request that also requests any Obligations.

**Example 6–6 Making PEP API Request and Parsing Response**

```java
Subject s = ...; // a Jps subject (with app roles inside)
String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
Map<String, String> env = new HashMap<String, String>();
env.put("myAttr", "Hello");

PepResponse response =
PepRequestFactoryImpl.getPepRequestFactory().newPepRequest
(s, action, resourceString, env).decide();
System.out.println("result: " + response.allowed());
Map<String, Obligation> obligations = response.getObligations();
for (String name : obligations.keySet())
{ System.out.print("obligation: name= " + name + ", value s="
obligations.get(name).getStringValues()); } 
```

Example 6–7 is an example of an Obligation output. Again, there are no double quotes around the string value.

**Example 6–7 Returned Obligations from Example 6–6**

```
result: true
obligation: name = MyObligation, values =
\{attr1=18, attr2=World, time=08:59:59, attr_date=12/29/2010\}
```

6.1.2.4 Making Java PEP API Bulk Authorization Requests

Example 6–8 is sample code written to request authorization in bulk. It encapsulates two requests.

**Example 6–8 Requesting Bulk Authorization with the Java PEP API**

```java
public void testBulkRequest() throws Exception {
Map<String, String> env = new HashMap<String, String>();
env.put("dynamic_attr", "dynamic_attr_value");
String resourceString =
MY_APPLICATION + "/" + MY_RESOURCE_TYPE + "/" + MY_RESOURCE;
String wrongAction = "wrong_action";
PepResponse resp = pepRequestFactory.newBulkPepRequest(
subject,
Arrays.asList(new Object[]{MY_ACTION, wrongAction}),
Arrays.asList(new Object[]{resourceString, resourceString}),
env).decide();
// response corresponding to first authorization decision
```

```
//
// assertTrue(resp.next());
// assertTrue("resp.allowed() is expected to be true!! ", resp.allowed());
// assertTrue("resp.allowed() is expected to be false!! ", resp.allowed());
//}
Example 6–10  Defining the New Java PEP API Response Type

```java
public PepRequest newQueryPepRequest(
    java.lang.Object subjectObj,
    java.lang.Object resourceObj,
    java.lang.Object actionObj,
    java.lang.Object environmentObj,
    org.openliberty.openaz.azapi.constants.PepResponseType responseType,
    boolean resourceActionOnly)
```

Example 6–11 is a query requesting all allowed and denied actions against a particular Resource and its children. Note that the search scope is defined as children.

Example 6–11  Verbose Query For a Resource and Children with Java PEP API

```java
String scope = "resource=PepQueryTest/Hierarchical/\res1";
PepRequest req = PepRequestFactoryImpl.getPepRequestFactory
    (subject, env, scope, PepRequestQueryType.VERBOSE);
PepResponse resp = req.decide();
ArrayList arraylist;
List grantedActions;
List deniedActions;

int i = 0;

//there can be more than 1 result when searchscope="children"
while (resp.next()) {
    RuntimeResource res = (RuntimeResource) resp.getResource();

    //both granted actions and denied actions are returned for
    //PepRequestQueryType.VERBOSE
    //PepResponse.getAction() returns an ArrayList where ArrayList.get(0) returns list
    //of granted actions;
    //it returns an ArrayList where ArrayList.get(1) returns list of denied actions;

    arraylist = (ArrayList) resp.getAction();
    grantedActions = null;
    deniedActions = null;

    if (arraylist != null) {
        grantedActions = (List) arraylist.get(0);
        deniedActions = (List) arraylist.get(1);
    }

    String resourceName = res.getResourceName();
}
```

Example 6–12 illustrates query code that returns only allowed results in the immediate search scope.

Example 6–12  Query Request for Immediate Searchscope with Java PEP API

```java
String scope = "resource=PepQueryTest/resource_type_1/resource_1,
    actions=action1, searchscope=immediate";
PepRequest req =
    PepRequestFactoryImpl.getPepRequestFactory().newQueryPepRequest
    (subject, env, scope, PepRequestQueryType.RETURN_ONLY_ALLOWED_RESULTS);
PepResponse resp = req.decide();
```
//Getting list of RuntimeAction objects
List actions = (List) resp.getAction();
RuntimeResource resource = (RuntimeResource) resp.getResource();

Example 6–13 illustrates query code that returns allowed and denied results in the
children search scope.

Example 6–13  Query Request for Children Searchscope with Java PEP API

String scope = "resource=PepQueryTest/Hierarchical/\\/res1";
PepRequest req = PepRequestFactoryImpl.getPepRequestFactory
  {subject, env, scope, PepRequestQueryType.VERBOSE};
PepResponse resp = req.decide();

ArrayList arrayList;
List grantedActions;
List deniedActions;

int i = 0;

//there can be more than 1 result when searchscope="children"
while (resp.next()) {
    RuntimeResource res = (RuntimeResource) resp.getResource();

    //both granted actions and denied actions are returned for
    //PepRequestQueryType.VERBOSE
    //PepResponse.getAction() returns an ArrayList where ArrayList.get(0)
    //returns list of granted actions;
    //it returns an ArrayList where ArrayList.get(1) returns list of denied actions;

    arrayList = (ArrayList) resp.getAction();
grantedActions = null;
deniedActions = null;

    if (arrayList != null) {
        grantedActions = (List) arrayList.get(0);
deniedActions = (List) arrayList.get(1);
    }
    String resourceName = res.getResourceName();
}

Example 6–14 illustrates query code that returns only allowed and denied results
along with any Obligations.

Example 6–14  Query Request for Allow/Deny Results & Obligations with Java PEP API

String resourceString = "MyApplication/MyResourceType/MyResource";
String action = "read";
PepRequest req = pepRequestFactory.newQueryPepRequest(subject,
  resourceString, action, null, PepResponseType.ALL_RESULTS, true);
PepResponse resp = req.decide();
while (resp.next()) {
    //print authorization results for each resource and action.
    System.out.println("For resource " + resp.getResource() + " and action" +
        resp.getAction() + " authorization allowed = " + resp.allowed());

    //check out obligations
    Map<String, Obligation> obligations = resp.getObligations();
    Set<String> obIds = obligations.keySet();
Using the PEP API

... for (String obId : obIds) {
    Obligation obligation = obligations.get(obId);
    Map<String, String> strValus = obligation.getStringValues();
}

6.1.2.6 Configuring the Java PEP API

To use the Java PEP API, details regarding the identity store, the policy store, the Policy Distribution Service, and the user assertion login module must be defined in the jps-config.xml configuration file.

**Note:** See Oracle Fusion Middleware Application Security Guide for more information on jps-config.xml configuration file. Parameters specific to Oracle Entitlements Server are documented in Oracle Fusion Middleware Administrator’s Guide for Oracle Entitlements Server.

Example 6–15 is a snippet of jps-config.xml, copied for informational purposes.

**Example 6–15 Sample jps-config.xml File**

```
...<serviceInstance name="idstore.ldap" provider="idstore.ldap.provider">
    <description>LDAP Identity Store Service Instance</description>
    <property name="idstore.config.provider" value="oracle.security.jps.wls.internal.idstore.WlsLdapIdStoreConfigProvider"/>
    <property name="CONNECTION_POOL_CLASS" value="oracle.security.idm.providers.stdldap.JNDIPool"/>
</serviceInstance>
<serviceInstance name="idstore.loginmodule" provider="jaas.login.provider">
    <description>Identity Store Login Module</description>
    <property name="loginModuleClassName" value="oracle.security.jps.internal.jaas.module.idstore.IdStoreLoginModule"/>
    <property name="jaas.login.controlFlag" value="REQUIRED"/>
</serviceInstance>
<serviceInstance name="pdp.service" provider="pdp.service.provider">
    <description>Runtime PDP service instance</description>
    <property name="oracle.security.jps.runtime.pd.client.sm_name" value="mixedsm"/>
    <property name="oracle.security.jps.runtime.pd.client.localpolicy.work_folder" value="/ade/diazhao_entsec_2/oracle/work/utp/buildout/functional/platsec/tempdir/mixedpd-work"/>
    <property name="oracle.security.jps.pdp.AuthorizationDecisionCacheEnabled" value="true"/>
    <property name="oracle.security.jps.pdp.AuthorizationDecisionCacheEvictionCapacity" value="500"/>
    <property name="oracle.security.jps.pdp.AuthorizationDecisionCacheEvictionPercentage" value="10"/>
    <property name="oracle.security.jps.pdp.AuthorizationDecisionCacheTTL" value="60"/>
    <property name="oracle.security.jps.pd.client.PollingTimerEnabled" value="true"/>
    <property name="oracle.security.jps.pd.client.PollingTimerInterval" value="10"/>
    <property name="oracle.security.jps.ldap.cache.refresh.interval" value="10000"/>
    <property name="oracle.security.jps.runtime.pd.client.policyDistributionMode" value="mixed"/>
</serviceInstance>
```
6.1.3 Using the PEP API for .NET

The PEP API for .NET can be used to integrate Oracle Entitlements Server authorization into applications built using Microsoft .NET technology. The .NET application calls the PEP API for .NET to get an authorization decision. (The PEP API for .NET invokes the authorization web service exposed by the Web Services Security Module through its proxy implementation (which handles the failover and caching).

To invoke the .NET Security Module for authorization, create an IPepRequest where a list of action objects and a corresponding list of resource objects are provided to represent applicable resource-action pairs. An authorization decision for each resource-action pair will be returned when IPepRequest.decide() is invoked. The following sections contain information on how to use the PEP API for .NET applications.

- Section 6.1.3.1, "Getting a .NET PEP API Subject"
- Section 6.1.3.2, "Making Simple .NET PEP API Authorization Requests"
- Section 6.1.3.3, "Processing .NET PEP API Obligations"
- Section 6.1.3.4, "Making .NET PEP API Bulk Authorization Requests"
- Section 6.1.3.5, "Making .NET PEP API Query Requests"
- Section 6.1.3.6, "Configuring the .NET PEP API"

6.1.3.1 Getting a .NET PEP API Subject

You can define a subject for the IPepRequest call by doing one of the following:

- Pass the user name as a String.
Pass the `System.Security.Principal.IIdentity` user object. This object is the .NET standard and signifies authentication by the Windows operating system which Oracle Entitlements Server supports. Get the `IIdentity` user object as follows:

- For a standalone application, use:
  ```csharp
  IIdentity iIdentity = WindowsIdentity.GetCurrent();
  ```

- For a web application, use:
  ```csharp
  IIdentity iIdentity = (IIdentity)(HttpContext.Current.User.Identity);
  ```

Use the `UserPrincipal` class to set the user information and pass a `UserPrincipal` object if the .NET application has custom authentication.

### 6.1.3.2 Making Simple .NET PEP API Authorization Requests

Oracle Entitlements Server offers requests for authorization (single and bulk) and queries. You can query for a list of all actions for a particular Resource (and its children), or you can request complete authorization results for a particular Resource (and its children). Both types of queries can also be used to retrieve results for all instantiated Resources of a given Resource Type.

Authorization information that is passed includes an object corresponding to authenticated subject (for example, `IIdentity`), a list of objects representing the actions (for example, `String {read}`), a list of objects representing the Resources (for example, `applicationId/resourcetype/resource`), and an object representing the environment (containing information passed in the form of a hashtable). See Section 6.1.4, "Formatting Authorization Request Strings" for information on how the scope and resource strings are specified in Example 6–16, "Requesting .NET PEP API Authorization Against a Resource". Note the code also contains an Obligation as discussed in Section 6.1.3.3, "Processing .NET PEP API Obligations."

#### Example 6–16 Requesting .NET PEP API Authorization Against a Resource

```csharp
String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
Hashtable env = new Hashtable();
env.Add("myAttr", "Hello");
IIdentity iIdentity = WindowsIdentity.GetCurrent();
Console.WriteLine("result: "+ response.Allowed());
```

#### Example 6–17 Requesting .NET PEP API Authorization with a String Subject

```csharp
String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
IPepResponse response = PepRequestFactoryImpl.GetPepRequestFactory().
```
NewPepRequest("John", action, resourceString).Decide();
Console.WriteLine("result: " + response.Allowed());

Example 6–18 requests authorization using the .NET PEP API when the subject is specified as UserPrincipal.

Example 6–18 Requesting .NET PEP API Authorization with a UserPrincipal Subject
UserPrincipal userPrincipal = new UserPrincipal("user1");
Hashtable envTable = new Hashtable();
String action = "read";
String resourceString = "MyApplication/MyResourceType/MyResource";
IPepResponse response = PepRequestFactoryImpl.GetPepRequestFactory().
    NewPepRequest(userPrincipal, action, resourceString, envTable).Decide();
Console.WriteLine("result: " + response.Allowed());

6.1.3.3 Processing .NET PEP API Obligations
The GetObligationId() method of the IObligation interface is used to retrieve any obligation relevant to the authorization request. An Obligation specifies optional information that is returned to the calling application with the access decision. Example 6–16 (in the previous section) is an authorization request that also gets any Obligations. Example 6–19 is another example.

Example 6–19 Requesting Single Authorization and Obligations with the .NET PEP API
String appId = "test-app";
String userName = "weblogic";
String resourceType = "Electronics-MyRT";
String resource1 = ":Nelco1";
String action1 = "view";
String action2 = "off";

IPepRequestFactory pepRequestFactoryImpl = PepRequestFactoryImpl.GetPepRequestFactory();
Hashtable envTable = new Hashtable();
envTable.Add("electronicType", "entertainment");
IPepResponse pepResponse = pepRequestFactoryImpl.NewPepRequest
    (userName, action1, appId + "/" + resourceType + "/" + resource1,
        envTable).Decide();
Dictionary<String,IObligation> obligations = pepResponse.GetObligations();
foreach (KeyValuePair<String, IObligation> entry in obligations)
{
    Console.WriteLine("Print the obligation" + entry.Value.GetStringValues());
}

6.1.3.4 Making .NET PEP API Bulk Authorization Requests
See Section 6.1.1.4, "Formatting Authorization Request Strings" for information on how the scope and resource strings are specified. Example 6–20 illustrates a call for bulk authorization.

Example 6–20 Requesting Bulk Authorization
String appId = "test-app";
String userName = "weblogic";
String authenticatedUser = "weblogic";
//String resourceType = "LibraryResourceType";
String resourceType = "Electronics-MyRT";
String resource1 = ":Nelco1";
String resource2 = ":Nelco1:TV";
String resource3 = ":Nelco1:TV:small-screen";
String action1 = "view";
String action2 = "on";
String action3 = "off";

IPepRequestFactory pepRequestFactoryImpl = PepRequestFactoryImpl.GetPepRequestFactory();
IPepResponse pepresponse=null;
bool isNext = false;
Collection<Object> resList = new Collection<Object>();
resList.Add(appId + "/" + resourceType + "/" + resource1);
resList.Add(appId + "/" + resourceType + "/" + resource2);
Collection<Object> actList = new Collection<Object>();
actList.Add(action1);
actList.Add(action3);
Hashtable envTable = new Hashtable();
envTable.Add("electronicType", "entertainment");
pepresponse = pepRequestFactoryImpl.NewBulkPepRequest
(userName, actList, resList, envTable).Decide();

//First authorization decision
isNext = pepresponse.Next();
Debug.Assert(isNext == false, "Error in bulk operation");
Debug.Assert(pepresponse.Allowed() == false, "Error in bulk operation");

//Second authorization decision
isNext = pepresponse.Next();
Debug.Assert(isNext == false, "Error in bulk operation");
Debug.Assert(pepresponse.Allowed() == true, "Error in bulk operation");

### 6.1.3.5 Making .NET PEP API Query Requests

The PEP API for .NET contains a newQueryPepRequest() method for retrieving information regarding the scope of resources and actions covered by the authorization request. The scope is based on subject and environment objects passed to the method. (See Section 6.1.1.4, "Formatting Authorization Request Strings" for information on how the scope and resource strings are specified.) The information returned depends on how the PepRequestQueryType is defined. It takes one of the following values.

- **RETURN_ONLY_ALLOWED_RESULTS** returns a list of resource action pairs that are allowed within the defined scope. See Example 6–21.
- **RETURN_ONLY_DENIED_RESULTS** returns a list of resource action pairs that are denied within the defined scope. See Example 6–22.
- **VERBOSE** returns all results for each resource action pairs requested within the defined scope. See Example 6–23.

#### Example 6–21 .NET Query for List of Allowed Resource Action Pairs

String appId = "MyApplication";
String resourceType = "resourceType";
String resource = "res";
IIdentity iIdentity = WindowsIdentity.GetCurrent();
IPepRequestFactory pepRequestFactoryImpl = PepRequestFactoryImpl.GetPepRequestFactory();
Using the PEP API

Handling Authorization Calls and Decisions

```java
IPepResponse pepresponse=null;
String actionStr="view,on,off";
bool isNext=false;
String scopeImmediate = "resource=" + "application" + "/" + "resource_type" + "/" + "resource2" + ",", actions=" + actionStr + ",searchscope=immediate";
Hashtable envTable = new Hashtable();
envTable.Add("electronicType", "entertainment");

pepresponse = pepRequestFactoryImpl.NewQueryPepRequest
  (iIdentity, envTable,scopeImmediate,PepRequestQueryType.
   RETURN_ONLY_ALLOWED_RESULTS).Decide();

isNext=pepresponse.Next();
Object act = pepresponse.GetAction();
List<String> actList = (List<String>)act;
if (actList.Contains("view") && actList.Contains("on"))
  {
    Debug.Assert(actList.Count == 2, "Invalid result");
  }
```

**Example 6–22 .NET Query for List of Denied Resource Action Pairs**

```java
String appId = "MyApplication";
String resourceType = "resourceType";
String resource = "res";
Identity iIdentity = WindowsIdentity.GetCurrent();
IPepRequestFactory pepRequestFactoryImpl =
  PepRequestFactoryImpl.GetPepRequestFactory();
IPepResponse pepresponse=null;
String actionStr = "view,on,off";
bool isNext = false;
String scopeImmediate = "resource=" + "application" + "/" + "resource_type" + "/" + "resource2" + ",", actions=" + actionStr + ",searchscope=immediate";
Hashtable envTable = new Hashtable();
envTable.Add("electronicType", "entertainment");
pepresponse = pepRequestFactoryImpl.NewQueryPepRequest
  (iIdentity, envTable,scopeImmediate,PepRequestQueryType.
   RETURN_ONLY_DENIED_RESULTS).Decide();

isNext = pepresponse.Next();
Object act = pepresponse.GetAction();
List<String> actList = (List<String>)act;
if (actList.Count == 1 || actList[0].Equals("off"))
  {
    Debug.Assert(actList.Count == 1, "Invalid result");
  }
```

**Example 6–23 .NET Query for List of All Resource Action Pairs**

```java
String appId = "MyApplication";
String resourceType = "resourceType";
String resource = "res";
Identity iIdentity = WindowsIdentity.GetCurrent();
//for standalone
IPepRequestFactory pepRequestFactoryImpl =
  PepRequestFactoryImpl.GetPepRequestFactory();
IPepResponse pepresponse=null;
String actionStr = "view,on,off";
String scopeImmediate = "resource=" + "application" + "/" + "resource_type" + "/" + "resource2" + ",", actions=" + actionStr + ",searchscope=immediate";
Hashtable envTable = new Hashtable();
```

Handling Authorization Calls and Decisions  6-19
envTable.Add("electronicType", "entertainment");

pepresponse = pepRequestFactoryImpl.NewQueryPepRequest
    (iIdentity, envTable, scopeImmediate, PepRequestQueryType.VERBOSE).Decide();

//isNext = pepresponse.Next();
Object act = pepresponse.GetAction();
List<List<String>> actList = (List<List<String>>)act;
List<String> actListAllowed = actList[0];
List<String> actListDenied = actList[1];
if (actListAllowed.Count == 2)
    { Debug.Assert(actListAllowed.Count == 2, "Invalid result"); }
if (actListDenied.Count == 1)
    { Debug.Assert(actListDenied.Count == 1, "Invalid result"); }

Example 6–24 illustrates a query for authorization results for a specified resource and its children. Note that searchscope=children.

Example 6–24 Requesting Authorization Results for a Resource and Its Children

IPepRequestFactory pepRequestFactoryImpl = PepRequestFactoryImpl.GetPepRequestFactory();
IPepResponse pepresponse=null;
String actionStr = "view,on,off";
String scopeImmediate = "resource=" + "application" + "/" + "resource_type" + 
    "/" + "resource2" + ", actions=" + actionStr + ",searchscope=children";
Hashtable envTable = new Hashtable();
envTable.Add("electronicType", "entertainment");

IIdentity iIdentity = WindowsIdentity.GetCurrent();
pepresponse = pepRequestFactoryImpl.NewQueryPepRequest
    (iIdentity, envTable, scopeImmediate, PepRequestQueryType.VERBOSE).Decide();

//isNext = pepresponse.Next();
Object act = pepresponse.GetAction();
List<List<String>> actList = (List<List<String>>)act;
List<String> actListAllowed = actList[0];
List<String> actListDenied = actList[1];
if (actListAllowed.Count == 2)
    { Debug.Assert(actListAllowed.Count == 2, "Invalid result"); }
if (actListDenied.Count == 1)
    { Debug.Assert(actListDenied.Count == 1, "Invalid result"); }

6.1.3.6 Configuring the .NET PEP API
The .NET PEP API are exposed to Microsoft .NET applications through Dynamic Link Libraries (DLL). The .NET application developer must incorporate OES-PEP.dll as part of the protected application. Also, as part of the configuration process:

1. Incorporate OES-PEP.dll as part of the protected application. This is done by the .NET application developer.

2. Update the following properties in the dotnetsm_config.properties file located in the OES_CLIENT_HOME/oessm/dotnetsm/configtool/ directory.
Making checkPermission() Calls

checkPermission() uses Java Permission objects to determine access to protected resources. A Java Permission object represents access to a resource and is constructed and assigned (access granted) based on the configured policy in effect. Oracle Entitlements Server supports the use of the checkPermission() method in the following standard classes:

- **gac.utility**: defines the Global Assembly utility path of the .NET framework with which the DLL have been registered.
- **wssm.smurl**: defines the URL of the Web Service Security Module used by the .NET application to communicate with Oracle Entitlements Server.
- **application.config.file**: defines the configuration file of the protected application. This parameter has two possible values. If it is a web application, the configuration file is named web.config; if it is a standalone application, the configuration file is named app.config.
- **operation**: defines the action for which you are preparing, this parameter has two possible values. config denotes configuration; remove will remove all configuration that has been added to the .NET application's configuration file as well as remove the registered OES-PEP.dll and log4net.dll from the GAC utility.
- **application.log4NetXmlfile**: defines the path of the log4net.xml which will generate the log file for the .NET Security Module. If the .NET application has already generated the log file, use the existing log4net.xml location. Otherwise, use the .NET Security Module's log4net.xml file located in OES_CLIENT_HOME/dotnetsm/logging/.

3. Run the configuration tool on the Windows machine using the following command where **smType** denotes the type of Security Module instantiated (in this case, dotnet) and **smConfigId** denotes the identifier defined when the Security Module was instantiated (in this case, myDotnet).

```bash
config.cmd -smType dotnet -smConfigId myDotnet
```

This tool will register OES-PEP.dll and log4net.dll on the Windows machine that is hosting the protected application and update the application's configuration file (web.config or app.config) with the locations of the Web Services Security Module and logging configuration files. Additionally, the following properties are defined:

- **oracle.security.jps.pdp.proxy.RequestTimeoutMilliSecs**: interval in which a request times out if the server is not responding. The default value is 10 (seconds).
- **oracle.security.jps.pdp.proxy.FailureRetryCount**: number of attempts to contact a server before trying an alternate failover server. The default value is 3 (attempts).
- **oracle.security.jps.pdp.proxy.FailbackTimeoutMilliSecs**: interval of time after which communication may be re-attempted with a primary server. The default value is 3 (minutes).
- **oracle.security.jps.pdp.proxy.SynchronizationIntervalMilliSecs**: defines how often the PDP Proxy polls the PDP server in order to synchronize state. For example, the interval is used to periodically check whether the authorization cache has to be flushed. The default value is 60 (seconds).

### 6.2 Making checkPermission() Calls

checkPermission() uses Java Permission objects to determine access to protected resources. A Java Permission object represents access to a resource and is constructed and assigned (access granted) based on the configured policy in effect. Oracle Entitlements Server supports the use of the checkPermission() method in the following standard classes:
This section discusses the use of the XACML Gateway in Oracle Entitlements Server. It explains how to use the `java.lang.SecurityManager` and `java.security.AccessController` classes. Additionally, it provides guidance on setting the `java.security.policy` system property and calling the `setPolicy()` method to explicitly set the policy provider.

### Note:
The static `AccessController.checkPermission` method uses the default access control context (the context inherited when the thread was created). To check permissions on some other context, call the instance `checkPermission()` method on a particular `AccessControlContext` instance.

Additionally, Oracle Entitlements Server supports the use of the `checkPermission()` method in the `oracle.security.jps.util.JpsAuth` class.

### Tip:
Oracle recommends the use of the `checkPermission()` method in the `oracle.security.jps.util.JpsAuth` class as it provides improved debugging support, better performance, and audit support.

When invoking the `checkPermission()` method (in a JavaSE application), make sure:

1. The `java.security.policy` system property has been set to the location of the Oracle Platform Security Services/Oracle WebLogic Server policy file.
2. Your application first calls the `setPolicy()` method to explicitly set the policy provider. This is illustrated by the following sample code.

```java
java.security.Policy.setPolicy(new
oracle.security.jps.internal.policystore.JavaPolicyProvider());
```

The `oracle.security.jps.util.JpsAuth.checkPermission()` works exactly as the standard methods by accepting a `Permission` object. If the requested access is allowed, `checkPermission()` returns quietly; if denied, an `AccessControlException` is thrown. Example 6–25 illustrates how you might use `checkPermission()`.

### Example 6–25 Using the checkPermission() Method

```java
java.security.Policy.setPolicy(new
oracle.security.jps.internal.policystore.JavaProvider()); // Java SE env only
PolicyContext.setContextID(TARGET_APP); // Java SE env only

// authorization runtime
Subject s = new Subject(); s.getPrincipals().add(new WLSUserImpl("wcai"));
s.setReadOnly();
JpsSubject.invokeAs(s, new PrivilegedAction<Object>() {
    public Object run() {
        FilePermission perm2 = new FilePermission("HARRY_PORTER", "read");
        psAuth.checkPermission(perm2);
        return null;
    }
});
```

### 6.3 Using the XACML Gateway

Oracle Entitlements Server allows external applications to ask authorization questions using the XACML 2.0 protocol. The Web Services Security Module contains a XACML gateway that allows it to receive XACML authorization requests and return XACML authorization responses. This capability is supported only when using the Multi-Protocol Security Module.
The Web Services Security Module XACML gateway acts as a remote PDP. It uses the standard XACML 2.0 context to convey authorization requests and responses between the PEP and the PDP. Here is the processing sequence for a XACML authorization request.

1. The PEP (application) establishes a session, authenticates a user and gets a valid token for the principal.

Example 6–26 illustrates how to establish the session and send a XACML 2.0 authorization request. Example 6–27 illustrates an alternative option.

**Example 6–26  Sample Code to Establish Session For XACML Gateway**

```java
setupSession();
request = createRequest();
try {
    resp = xacmlSvc.authorize(request);
} catch (AxisFault af) {
    if (isTokenExpired(af)) {
        resetupSession();
        try {
            resp = xacmlSvc.authorize(request);
        } catch (RemoteException e) {
            throw new XACMLException("Error calling the XACML service.", e);
        }
    } else {
        throw new XACMLException("Error calling the XACML service.", af);
    }
} catch (RemoteException e) {
    throw new XACMLException("Error calling the XACML service.", e);
}

private boolean isTokenExpired(AxisFault af) {
    String faultReason = af.getFaultReason();
    if((faultReason != null) && (faultReason.indexOf("IdentityAssertionException") != -1)) {
        return true;
    }
    return false;
}

private void setupSession() throws XACMLException {
    if (identity == null) {
        establishSession();
    }
}

private void resetupSession() throws XACMLException {
    establishSession();
}

private void establishSession() throws XACMLException {
    try {
        EstablishSessionType sess = new EstablishSessionType();
        sess.setPrincipalsInfo(convertSubjectToPrincipalsInfo(subject));
        sess.setRequestCredentialType(OES_CREDENTIAL_TYPE);
        AuthenticationResultType result = atzSvc.establishSession(sess);
        identity = result.getIdentityAssertion();
    }
```
Using the XACML Gateway

Example 6–27 illustrates a new mode which can pass the Subject on every request and doesn’t require the establishSession method. The previous mode, Example 6–26, is still supported.

Example 6–27 Sample Code To Establish Session with Principal Information

```java
/**
 * Test authorize for user role in new mode which send the principal info in every request.
 * @throws Exception
 */
public void testAtzForUserRoleNewMode() throws Exception {
    SubjectType sbjct1 = constructSubject(OTHER_USER, new String[]{OTHER_ROLE});
    SubjectType sbjct2 = constructSubject(ADMIN_USER, new String[]{ADMIN_ROLE});

    printTestHeader("Deny Test: user \\
" + OTHER_USER + "\n is in role \\
" +
    OTHER_ROLE + \\"");
    sendRequest(new SubjectType[]{sbjct1}, new ResourceType[]{resTest1},
        actionWrite, emptyEnv);

    printTestHeader("Permit Test: user \\
" + ADMIN_USER + "\n is in role \\
" +
    ADMIN_ROLE + \\"");
    sendRequest(new SubjectType[]{sbjct2}, new ResourceType[]{resTest1},
        actionWrite, emptyEnv);
}
```

2. The PEP sends a XACML request containing the token to the PDP (Security Module).

Example 6–28 illustrates how to create a XACML authorization request.

Example 6–28 Creating a XACML Request

```java
private RequestType createRequest() throws XACMLException {
    // create resource
    String res = "Library/LibraryResourceType/Book";
    AttributeType attr = createAttribute(res, RESOURCE_ID, XML_STRING_TYPE);
    ResourceType resource = new ResourceType(null, new AttributeType[]{attr});

    // create action
    String actionStr = "borrow";
    attr = createAttribute(actionStr, ACTION_ID, XML_STRING_TYPE);
    ActionType action = new ActionType(new AttributeType[]{attr});

    // create environment
    String isRegistered = input.getString("Is the user registered in the library (yes|no): ");
    String numberOfBorrowedBooks = input.getString("How many books has the user borrowed already:: ");
    EnvironmentType env;
```
List attrs = new ArrayList();
    attrs.add(createAttribute(isRegistered, XACML_NAMESPACE + "RegisteredAttribute", XML_STRING_TYPE));
    attrs.add(createAttribute(numberOfBorrowedBooks, XACML_NAMESPACE + "NumberOfBorrowedBooksAttribute", XML_STRING_TYPE));
    // obligations
    attrs.add(createAttribute(LIST_VAL1, XACML_NAMESPACE + ATTRIBUTE_NAME, XML_STRING_TYPE));
    attrs.add(createAttribute(LIST_VAL2, XACML_NAMESPACE + ATTRIBUTE_NAME, XML_STRING_TYPE));
    env = new EnvironmentType((AttributeType[])attrs.toArray(new AttributeType[attrs.size()]));
    // subject
    attr = createAttribute(identity.get_any(), SUBJECT_ID, XACML_NAMESPACE + OES_CREDENTIAL_TYPE);
    SubjectType subject = new SubjectType(new AttributeType[]{attr}, null);
    // now construct the request with subject, resource, action and environment.
    return new RequestType(new SubjectType[]{subject}, new ResourceType[]{resource}, action, env);
}

Example 6–29 is a sample XACML 2.0 authorization request. The SSM-SOAPWS_xacml.wsdl file provides the operation interface definitions.

Example 6–29 XACML 2.0 Authorization Request

```xml
<Request xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
    <Subject xsi:type="ns1:SubjectType"
        xmlns:ns1="urn:oasis:names:tc:xacml:2.0:context:schema:os"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
            xsi:type="ns1:AttributeType">
            <AttributeValue xsi:type="ns1:AttributeValueType">
                <OESIdentityAssertion
                    xmlns="http://security.bea.com/ssmws/ssm-soap-types-1.0.xsd">
                    SU=John;TS=1288702235781;CT=1</OESIdentityAssertion>
            </AttributeValue>
        </Attribute>
    </Subject>
    <ns2:Resource xsi:type="ns2:ResourceType"
        xmlns:ns2="urn:oasis:names:tc:xacml:2.0:context:schema:os"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <ns2:Attribute AttributeId="urn:oasis:names:tc:xacml:2.0:resource:resource-id"
            xsi:type="ns2:AttributeType">
            <ns2:AttributeValue xsi:type="ns2:AttributeValueType">
                Library/LibraryResourceType/Book</ns2:AttributeValue>
        </ns2:Attribute>
    </ns2:Resource>
    <ns3:Action xsi:type="ns3:ActionType"
        xmlns:ns3="urn:oasis:names:tc:xacml:2.0:context:schema:os"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <ns3:AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
            xsi:type="ns3:AttributeType">
            <ns3:AttributeValue xsi:type="ns3:AttributeValueType">borrow</ns3:AttributeValue>
        </ns3:Attribute>
    </ns3:Action>
</Request>
```
3. The XACML gateway asserts the token and converts it to the applicable identity.

4. Oracle Entitlements Server reaches an authorization decision regarding the principal using any applicable policies and returns a XACML response to the PEP.

Example 6–30 is a sample XACML 2.0 authorization response. The SSM-SOAPWS_xacml.wsdl file provides the operation interface definitions.

Example 6–30  XACML 2.0 Authorization Response

```xml
<Response xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
  <Result ResourceId="Library/LibraryResourceType/Book">
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
    <ns1:Obligations xmlns:ns1="urn:oasis:names:tc:xacml:2.0:policy:schema:os">
      <ns1:Obligation ObligationId="http://security.bea.com/ssmws/ssm-ws-1.0.wsd1#Roles" FulfillOn="Permit">
        <ns1:AttributeAssignment
          DataType="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="http://security.bea.com/ssmws/ssm-ws-1.0.wsd1#role">
          AuthenticatedUser
        </ns1:AttributeAssignment>
      </ns1:Obligation>
      <ns1:Obligation ObligationId="http://security.bea.com/ssmws/ssm-ws-1.0.wsd1#ResponseAttributes" FulfillOn="Permit">
        <ns1:AttributeAssignment
          DataType="http://www.w3.org/2001/XMLSchema#dateTime"
          AttributeId="http://security.bea.com/ssmws/ssm-ws-1.0.wsd1#decisionTime">
          2010-11-02T12:50:43.685Z
        </ns1:AttributeAssignment>
      </ns1:Obligation>
    </ns1:Obligations>
  </Result>
</Response>
```
An extension class can be loaded by the Oracle Entitlements Server runtime environment to enhance core functionality. Extensions are bundled as Java Archive (JAR) files. This chapter contains the following sections on extensions that can be created.

- Section 7.1, "Working With Attribute Retrievers"
- Section 7.2, "Developing Custom Functions"

### 7.1 Working With Attribute Retrievers

The Policy Information Point (PIP) is a system entity that acts as a source for attribute values. During runtime evaluation of a policy, Oracle Entitlements Server relies on an Attribute Retriever plug-in to get attribute values from one or more PIP information stores. These Attribute Retrievers allow policies to be data-driven in that the value of the attribute can impact the access decision. For example, if access to transfer money from a bank account is based on how much money is currently in the account, an Attribute Retriever can be used to get a value for the current balance. This infrastructure is highly extensible, allowing users to develop their own PIP plug-ins to retrieve information from many places - for example, from a file, a USB driver, or the internet.

**Note:** See Oracle Fusion Middleware Administrator’s Guide for Oracle Entitlements Server for a detailed explanation of the PIP.

The following sections have more information.

- Section 7.1.1, "Understanding Attribute Retrievers"
- Section 7.1.2, "Creating Custom Attribute Retrievers"
- Section 7.1.3, "Implementing Custom Attribute Retrievers"
- Section 7.1.4, "Configuring Oracle Entitlements Server for Custom Attribute Retrievers"

### 7.1.1 Understanding Attribute Retrievers

Oracle Entitlements Server uses predefined Attribute Retrievers to connect to Lightweight Directory Access Protocol (LDAP) data stores and relational database management systems (RDBMS). Custom Attribute Retrievers can be developed to get attribute values from other types of PIP data stores. A custom Attribute Retriever can return values for one or many attributes.
Configuration information for Attribute Retrievers is defined in the \texttt{jps-config.xml} configuration file. Configuration of the Attribute Retriever within this file is dependent on whether it is predefined or custom.

- For predefined Attribute Retrievers:
  - Configure information needed to connect to the data store as well as credential information.
  - Configure individual attribute values including attribute name, name of Attribute Retriever used, search query to retrieve the value (for example, SQL query if the PIP is a relational database or LDAP query if it's a directory), and any attribute value caching information).

  For detailed information on configuring predefined Attribute Retrievers, see the \textit{Oracle Fusion Middleware Administrator's Guide for Oracle Entitlements Server}.

- For custom Attribute Retrievers, configure information regarding the name of the class implementing the Attribute Retriever.

  A given Attribute Retriever can return a single value or multiple values attribute.

\textbf{Note:} See the \textit{Oracle Fusion Middleware Application Security Guide} for more information on the \texttt{jps-config.xml} configuration file. Parameters specific to Oracle Entitlements Server are documented in the \textit{Oracle Fusion Middleware Administrator's Guide for Oracle Entitlements Server}.

### 7.1.2 Creating Custom Attribute Retrievers

As described in \textit{Section 1.3.3, "Adding a Condition,"} a policy Condition is built using attributes or functions. If a dynamic attribute is used in a Condition, the attribute value can be passed in from the \texttt{com.bea.security.AppContext} interface or retrieved with either a predefined or custom Attribute Retriever. The following procedure documents the steps to create a custom Attribute Retriever.

1. Implement the custom Attribute Retriever using the \texttt{com.bea.security.providers.authorization.asi.AttributeRetrieverV2} interface.

   See \textit{Section 7.1.3, "Implementing Custom Attribute Retrievers"} for more information.

2. Create a JAR file.

3. Add the JAR file to the appropriate classpath.
   - If connecting to a Java Security Module, add the JAR file to the application classpath.
   - If connecting to an RMI, Web Services or WebLogic Server Security Module, add the JAR file to the system classpath with the rest of the Security Module JAR files.

   It does not matter where the JAR is physically stored.

4. Configure the Security Module to use the custom Attribute Retriever.

   Make sure the configuration specifies the fully-qualified location of the custom Attribute Retriever. For more information, see \textit{Oracle Fusion Middleware Administrator's Guide for Oracle Entitlements Server}. 

---

7-2  

\textit{Developer's Guide for Oracle Entitlements Server}
7.1.3 Implementing Custom Attribute Retrievers

A custom Attribute Retriever must implement the AttributeRetrieverV2 interface. Table 7–1 explains the methods available for this purpose.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAttributeValue()</td>
<td>This method is called every time the value of a particular attribute is required. It returns the value of the named attribute and takes the following parameters:</td>
</tr>
<tr>
<td></td>
<td>- Name defines the name of the attribute being retrieved.</td>
</tr>
<tr>
<td></td>
<td>- RequestHandle is the interface that will retrieve values of other attributes (if required). It also allows the sharing of context – an arbitrary Object - between different attribute retrievers or custom functions.</td>
</tr>
<tr>
<td></td>
<td>- Subject defines the user associated with the request.</td>
</tr>
<tr>
<td></td>
<td>- Roles defines any role membership of the subject, or null if this is a role mapping call.</td>
</tr>
<tr>
<td></td>
<td>- Resource defines the protected resource associated with the request.</td>
</tr>
<tr>
<td></td>
<td>- contextHandler is the context associated with the request; this may be null if non-existent.</td>
</tr>
<tr>
<td>getHandledAttributeNames()</td>
<td>This method is called once, usually during loading of the attribute retriever. The method returns the list of attribute names for which the attribute retriever can return values.</td>
</tr>
</tbody>
</table>

In the simplest use case, an Attribute Retriever does not need additional information to get the value. For example, to get the time of day, getAttributeValue() calls a system function and returns the information. Another use case might find the Attribute Retriever needs additional information before it can return an attribute value. For example, the Attribute Retriever would have to know the user’s identifier in order to get the location of the user. For this purpose, the Attribute Retriever is provided a RequestHandle interface to get the values for other attributes. In this example, the attribute retriever can use the RequestHandle interface to get the value of the built-in SYS_USER attribute which resolves to the identity of the current user.

**Note:** Names of system attributes must be placed between percentage (%) signs as in %sys_user%.

The following sections contain more information on the attribute retrieval options.

- Section 7.1.3.1, "Getting Attribute Values Directly"
- Section 7.1.3.2, "Getting Attribute Values Using a Handle"

### 7.1.3.1 Getting Attribute Values Directly

An implementation of AttributeRetrieverV2 can use the getAttributeValue() method to return the value of a named attribute. This method takes as input the name of the attribute whose value will be returned. Example 7–1 illustrates how getAttributeValue() might be used.

#### Example 7–1 Implementing getAttributeValue() Method

```java
package oracle.security.oes.test;
```
import java.util.Map;
import javax.security.auth.Subject;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.Resource;
import com.bea.security.providers.authorization.asi.AttributeRetrieverV2;
import com.bea.security.providers.authorization.asi.ARME.evaluator.RequestHandle;

public class SimpleAttributeRetriever implements AttributeRetrieverV2 {
    public Object getAttributeValue(String name, RequestHandle requestHandle,
        Subject subject, Map roles, Resource resource,
        ContextHandler contextHandler) {
        if (name == null) return null;
        return "static_value";
    }

    public String[] getHandledAttributeNames() {
        return new String[] {"static_attr"};
    }
}

MyAttributeRetrieverV2 is the implementation of AttributeRetrieverV2. The
getHandledAttributeNames() method returns the names of attributes handled by this
implementation. It may return at least one attribute name; an empty or null value
indicates that the retriever will be called for any attribute. The values of the
getAttributeValue() parameters are defined as:

- name is the name of the attribute being retrieved.
- requestHandle is the implementation of the interface that allows you to retrieve
  values of other attributes (if required). It also allows the sharing of context – an
  arbitrary Object - between different attribute retrievers or custom functions. It is
  passed to the function even if it is not used.
- subject is the principal associated with the request.
- roles defines the role membership of the associated principal. The object is a map
  where the key signifies the role name and the value is the role object.
- resource is the protected resource associated with the request.
- contextHandler defines the context associated with the request. It may be null if
  the context is non-existent.

7.1.3.2 Getting Attribute Values Using a Handle
In some cases, the Attribute Retriever might need to get an attribute for information
before retrieving the attribute value it wants. For example, in order to get the location
of a user, the attribute retriever would need the identifier of the user. By invoking the
getAttribute() method in the RequestHandle interface, the Attribute Retriever is able
to get the identifier and with it access to all of the user’s information. The
getAttribute() method returns the attribute name and value as a name-value pair in
an AttributeElement object.
RequestHandle is the interface that allows you to retrieve values of other attributes if required. It also allows to share context – arbitrary Object - between different invocation of Attribute Retrievers and/or custom functions.

Note: The getAttribute() method is used to retrieve values for user and resource attributes. It should not be used to get values for dynamic or extension attributes.

Example 7–2 illustrates how getAttribute() might be used.

Example 7–2 Using getAttribute() Method

```java
public Object getAttributeValue
    (String name, RequestHandle requestHandle, Subject subject,
     Map roles, Resource resource, ContextHandler contextHandler) {

    ... ...

    // retrieve sys_user built-in attribute
    String user = null;
    try {
        AttributeElement element = requestHandle.getAttribute("sys_user", true);
        if (element != null) {
            user = (String)element.getValueAs(String.class);
        }
    } catch (Exception e) {
        // ignore it
    }

    ... ...
}
```

The values of the getAttribute() parameters are defined as:

- sys_user is the name of the attribute being retrieved.
- true enables the attribute type check functionality. The value may be false to disable the type check.

7.1.4 Configuring Oracle Entitlements Server for Custom Attribute Retrievers

This section contains the procedure on how to configure Oracle Entitlements Server to recognize a custom attribute retriever. After implementing `com.bea.security.providers.authorization.asi.AttributeRetrieverV2` (as discussed in Section 7.1.3, "Implementing Custom Attribute Retrievers"), compile the Java code, add the compiled class to the class path of the Security Module instance, and make the following changes to the `jps-config.xml` configuration file.

1. Declare the PIP service provider in the `<serviceProviders>` section as illustrated in Example 7–3.

Example 7–3  serviceProviders Section of jps-config.xml

```xml
<serviceProviders>
    <serviceProvider
        class="oracle.security.jps.az.internal.runtime.provider.PIPServiceProvider"
        name="pip.service.provider" type="PIP"/>
</serviceProviders>
```
2. Declare the PIP service instance in the `<serviceInstances>` section as illustrated in Example 7–4.

Example 7–4 serviceProviders Section of jps-config.xml

```xml
<serviceInstances>
  <serviceInstance name="pip.service.MyAttributeRetriever"
                   provider="pip.service.provider">
    <property name="type" value="CUSTOM_PIP"/>
    <property name="application" value="testPIPBasedOnCustomPIP"/>
    <property name="description" value="MyAttributeRetriever"/>
    <property name="classnames" value="pips.MyDummyAttributeRetriever"/>
  </serviceInstance>
</serviceInstances>
```

Specify the properties as defined in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>The value should be CUSTOM_PIP</td>
</tr>
<tr>
<td>Application</td>
<td>The application in which the PIP instance takes effect</td>
</tr>
<tr>
<td>Classnames</td>
<td>The fully qualified class name of the custom PIP</td>
</tr>
</tbody>
</table>

3. Declare the PIP service instance in the `<jpsContext>` section as illustrated in Example 7–5.

Example 7–5 jpsContext Section of jps-config.xml

```xml
<jpsContext name="default">
  <serviceInstanceRef ref="pip.service.MyAttributeRetriever"/>
</jpsContext>
```

7.2 Developing Custom Functions

A function can be used in a policy Condition to perform some advanced operation. The function may have a number of parameters and can return any of the supported data types. Oracle Entitlements Server provides a number of predefined functions and, additionally, allows you to declare your own.

7.2.1 Implementing a Custom Function

The following procedure details the steps to take when implementing a custom function in your policy.

1. Write the custom code for the function. See Example 7–6.
2. Compile the Java code and create a jar file.
3. Add the compiled class to the class path.
4. Restart the servers.

7.2.1.1 Step 1 Write the Custom Code

A custom function can be implemented as a method in a class that may contain one or more custom functions. You can choose any method name as long as the name...
matches the corresponding name referenced in the policy. Custom functions can be
passed as arguments consisting of constants or names of other attributes (including
dynamic attributes) or names of other functions. Since all evaluation functions share a
common namespace, two functions cannot have the same name.

Example 7–6 illustrates how you might create a custom function.

**Example 7–6 Sample Java Pseudo Code for a Custom Function**

```java
//Package declaration
package oracle.security.oes.extensions;

import java.util.Map;

import javax.security.auth.Subject;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.Resource;
import com.bea.security.providers.authorization.asi.ARM.E.evaluator.RequestHandle;
import com.wles.util.AttributeElement;

public class MyCustomFunction {

    /**
     * Named evaluation function. Additional authorization request data
     * is made available to allow for more complex attribute evaluation.
     * This method will be registered to an application and be invoked while
     * the policy contains a custom evaluation function with the name
     * "my_custom_function".
     *
     * @param requestHandle an attributes container associated with the request,
     * through which the function can get required attribute value.
     *
     * @param args an array of function arguments. Each element is either
     * <code>null</code> or a String
     *
     * @param subject the subject associated with the request
     *
     * @param roles the role membership map of the subject
     * key: role name.
     * value: role object <code>null</code> if function
     * is called during role mapping
     *
     * @param resource the resource associated with the request
     *
     * @param contextHandler the context associated with the request, may be
     * <code>null</code> if non-existent
     *
     * @return <Return_data_type> as the result of the function
     *
     * @throws Exception if the function cannot get required attribute value.
     */

    public <Return_data_type> my_custom_function(RequestHandle requestHandle,
                                               Object[] args,
                                               Subject subject,
                                               Map roles,
                                               ...
Developing Custom Functions

Resource resource,
ContextHandler contextHandler) {

    // Check if we have a correct number of the input parameters
    if(args.length < <required_number_of_attributes> ||
       args[0] == null || args[1] == null) {
        // Incorrect number of arguments.
        // Such a policy is invalid and cannot be evaluated
        throw new RuntimeException
          ("Incorrect number of arguments provided to the function");
    }

    // Arguments for an evaluation function are attribute names.
    // If a string literal or a numeric value is used, then
    // it is passed in as value. The only way to distinguish
    // values from names is to try to look up the attribute.
    // Evaluation function should not set any values.
    try {
        // The body of the custom function.
        // It should return the value of the type specified in the method
        // signature.

    } catch(Exception e) {
        // Catch the exception and throw with error message
        throw new RuntimeException(
          "Detailed error message .... " + " Exception: " +
          e.getMessage());
    }

    return <return_data_type_value>;
}

7.2.1.2 Step 2. Compile the Source Code and Include the Class in a JAR File
To compile the source code:
1. Set JAVA_HOME and PATH based on your JVM location.
2. Set your Classpath to include the OES jar file to be included in the compilation.
3. Compile the code and create a JAR file called custom_funcs.jar.

7.2.1.3 Step 3. Add the Jar File to CLASSPATH of WebLogic Domain on which the Application is Deployed
To add a JAR file to the classpath:
1. Make a copy of setDomainEnv.sh file.
2. Add the Jar file to POST_CLASSPATH.
   POST_CLASSPATH="${POST_CLASSPATH}${CLASSPATHSEP}${ORACLE_HOME}/dist/custom_funcs.jar"
   export POST_CLASSPATH

7.2.1.4 Step 4. Restart the Servers
Restart the servers.
7.2.2 Using InspectableFunction For Metadata Information

An argument metadata interface is used to query for information about the arguments expected by a custom Oracle Entitlements Server function. (This includes information such as the number of arguments and their names and types.) Optionally, the InspectableFunction interface can be implemented for these argument queries. When adding a custom function based on this interface, the Administration Console calls the getArgMetadata method to return metadata describing the arguments, and verify the metadata that is expected by the custom function.

For each argument expected by the function starting with the first (argument 0), getArgMetadata must return metadata describing the argument. Once the Administration Console collects the metadata about each expected argument, it uses the interface methods getArgValue and isValidArgValue to automate the process of working with custom Oracle Entitlements Server functions.

- Example 7–7 illustrates sample code for the getArgMetadata(String functionName) method.
- Example 7–8 illustrates sample code for the getArgValues (String functionName, int argNumber, ArrayList<oracle.security.jps.service.policystore.info.DataType> prevArgValues) method.
- Example 7–9 illustrates sample code for the isValidArgValue (String functionName, int argNumber, oracle.security.jps.service.policystore.info.DataType argValue, ArrayList<oracle.security.jps.service.policystore.info.DataType> prevArgValues) method.

Example 7–7  Sample Code for getArgMetadata() Method

```java
package InspectableFunctionSample;

import com.bea.security.providers.authorization.asi.ARME.evaluator.RequestHandle;
import com.bea.security.providers.authorization.asi.InspectableFunction;
import java.io.PrintStream;
import java.util.*;
import javax.security.auth.Subject;
import oracle.security.jps.service.policystore.info.*;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.Resource;

/**
 * Custom OES function to validate an address. The number indicates the order used to emulate
display and selection process.
 * arg[0]: country, arg[1]: city, arg[2]: street, arg[3]: house number, arg[4]: mailbox
 * entering previous values will help to provide options for late values
 */

public class Address
    implements InspectableFunction
{

    public Address()
    {
    }

    public boolean validateAddress(RequestHandle requestHandle, Object args[], Subject subject, Map roles, Resource resource, ContextHandler contextHandler)
        throws RuntimeException
    {
```

Example 7–8 illustrates sample code for the getArgValues method.
Developing Custom Functions

7-10  Developer's Guide for Oracle Entitlements Server

```java
public boolean validateAddress(String country, String city, String street, int house, boolean mailbox) {
    int argsLen = args.length;
        throw new RuntimeException("Incorrect number of arguments in a function");
    } else {
        String country = args[0].toString();
        String city = args[1].toString();
        String street = args[2].toString();
        int house = Integer.parseInt(args[3].toString());
        boolean mailbox = Boolean.parseBoolean(args[4].toString());
        boolean result = country.compareToIgnoreCase("USA") == 0 &&
                         city.compareToIgnoreCase("Boston") == 0 &&
                         street.compareToIgnoreCase("Beacon Str") ==
                           0 && house == 10 && mailbox;
        System.out.println((new StringBuilder()).append("Result = " + result).append(" for arguments Country = ",
                                          city).append(" , City = ",
                                          street).append(" , house = ",
                                          house).append(" , mailbox = ").toString());
        return result;
    }
}

public ArrayList getArgMetadata(String functionName) {
    if (functionName.compareToIgnoreCase("validateAddress") == 0) {
        ArrayList metadata = new ArrayList();
        metadata.add(new com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata(0, "Country",
                                                                                             OpssString.class, false, true, true));
        metadata.add(new com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata(1, "City",
                                                                                             OpssString.class, false, true, true));
        metadata.add(new com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata(2, "Street",
                                                                                             OpssString.class, false, true, true));
        metadata.add(new com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata(3, "House",
                                                                                             OpssInteger.class, true, false, false));
                                                                                             OpssBoolean.class, false, true, false));
        return metadata;
    } else {
        throw new RuntimeException("Invalid function name");
    }
}

public ArrayList getArgValues(String functionName, int argNumber, ArrayList prevArgValues) throws RuntimeException {
    ArrayList values = new ArrayList();
    if (functionName.compareToIgnoreCase("validateAddress") == 0) {
        if (argNumber == 0) {
            values.add(new OpssString("USA"));
            values.add(new OpssString("Canada"));
        } else
```
if(argNumber == 1)
{
    if(((DataType)prevArgValues.get(0)).toString().compareToIgnoreCase("USA")
    == 0)
    {
        values.add(new OpssString("Boston");
        values.add(new OpssString("San Francisco");
    } else
    if(((DataType)prevArgValues.get(0)).toString().compareToIgnoreCase("Canada")
    == 0)
    {
        values.add(new OpssString("Montreal");
        values.add(new OpssString("Toronto");
    } else
    {
        throw new RuntimeException("Invalid argument value");
    }
} else
if(argNumber == 2)
{
    if(((DataType)prevArgValues.get(0)).toString().compareToIgnoreCase("USA")
    == 0)
    {
        if(((DataType)prevArgValues.get(1)).toString().compareToIgnoreCase("Boston") == 0)
        {
            values.add(new OpssString("Beacon Str");
            values.add(new OpssString("Boylston Str");
        } else
        if(((DataType)prevArgValues.get(1)).toString().compareToIgnoreCase("San Francisco") == 0)
        {
            values.add(new OpssString("Van Ness Str");
            values.add(new OpssString("Market Str");
        } else
        {
            throw new RuntimeException("Invalid USA city");
        }
    } else
    if(((DataType)prevArgValues.get(0)).toString().compareToIgnoreCase("Canada")
    == 0)
    {
        if(((DataType)prevArgValues.get(1)).toString().compareToIgnoreCase("Montreal") == 0)
        {
            values.add(new OpssString("St. Laurent Str");
            values.add(new OpssString("St. Catherine Str");
        } else
        if(((DataType)prevArgValues.get(1)).toString().compareToIgnoreCase("Toronto") == 0)
        {
            values.add(new OpssString("Queen Str");
            values.add(new OpssString("King Str");
        } else
        {
            throw new RuntimeException("Invalid Canada city");
        }
    } else
    {
        throw new RuntimeException("Invalid country");
    }
} else
if(argNumber == 3)
{
    values.add(new OpssInteger(10));
    values.add(new OpssInteger(15));
Developing Custom Functions

```java
7-12

Execute code here to develop custom functions.

```
if(((DataType)prevArgValues.get(1)).toString().compareToIgnoreCase("Toronto") == 0)
    return argValue.toString().compareToIgnoreCase("Queen Str") == 0 ||
    argValue.toString().compareToIgnoreCase("King Str") == 0;
else
    throw new RuntimeException("Invalid Canada city");
}
else{
    throw new RuntimeException("Invalid country");
}
if(argNumber == 3 && argValue.getClass() ==
    OpssInteger.class)
    return true;
if(argNumber == 4 && argValue.getClass() ==
    OpssBoolean.class)
    { 
    if(((OpssInteger)prevArgValues.get(3)).intValue() == 10)
        return !((OpssBoolean)argValue).booleanValue();
    if(((OpssInteger)prevArgValues.get(3)).intValue() == 15)
        return ((OpssBoolean)argValue).booleanValue();
    else
        return false;
    }
else
    throw new RuntimeException("Invalid argument number or value type");
}
else
    throw new RuntimeException("Invalid function name");
}

public static void main(String args[])
{
    System.out.println("*** Inspecting function 'validateAddress' ***");
    ArrayList metadataList = OES_FUNCTION_CLASS.getArgMetadata("validateAddress");
    System.out.println(new StringBuilder().append("\nNumber of arguments : ").append(metadataList.size()).toString());
    com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata metadata;
    for(Iterator i$ = metadataList.iterator(); i$.hasNext(); System.out.println((new StringBuilder()).append("Arg ").append(metadata.getArgNumber()).append(" : name = ").append(metadata.getArgName()).append(" , class = ").append(metadata.getArgClass()).append(" , manual entry = ").append(metadata.isArgAllowManualEntry()).append(" , dependent = ").append(metadata.isArgDependent()).append(" , clear dependent args = ").append(metadata.isArgOnChangeClearDependentArgs()).append(" , toString() = ").toString()))
        metadata = (com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata)i$.next();

    System.out.println("\n*** Testing address selection for USA ***");
    ArrayList prevArgValuesUsa = new ArrayList();
    apm(0, prevArgValuesUsa, 0);
    apm(1, prevArgValuesUsa, 1);
    apm(2, prevArgValuesUsa, 0);
    apm(3, prevArgValuesUsa, 1);
    apm(4, prevArgValuesUsa, 0);
    System.out.println("\nFinal selection :");
    for(int i = 0; i < 5; i++)
        System.out.println((new StringBuilder()).append(" - ").append(((com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata)metadataList.get(i)).getArgName()).append(" = ").append(((DataType)prevArgValuesUsa.get(i)).toString()).toString());
    System.out.println(new StringBuilder().append("\nExpecting Not Valid...").append(((com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata)metadataList.get(5)).getArgName()).append(" = ").append(((DataType)prevArgValuesUsa.get(5)).toString()).toString());
Developing Custom Functions

```java
private static void apm(int argnum, ArrayList prevArgValues, int selection) {
    ArrayList metadataList = OES_FUNCTION_CLASS.getArgMetadata("validateAddress");
    System.out.println(new StringBuilder().append("\n").append("\nGet values for argument '").append(((com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata)metadataList.get(argnum)).getArgName()).append("'").toString());
    ArrayList argValues = OES_FUNCTION_CLASS.getArgValues("validateAddress", argnum, prevArgValues); // Create an ArrayList of Argument Values
    DataType value = new OpssBoolean(false);
    for(int i = 0; i < 5; i++) {
        System.out.println((new StringBuilder()).append("- ").append(((com.bea.security.providers.authorization.asi.InspectableFunction.ArgMetadata)metadataList.get(i)).getArgName()).append(" = ").append(((DataType)prevArgValues.get(i)).toString()).toString());
    }
}
```

This code snippet demonstrates how to develop custom functions within the Oracle Entitlements Server. It involves creating an ArrayList of Argument Values and iterating through them to check the validity of address components such as city, street, and house numbers for both U.S. and Canada addresses. The `apm` function is used to print the values of arguments and to check their validity based on predefined conditions.
Developing Custom Functions

Extending Functionality

```java
for(Iterator i$ = argValues.iterator(); i$.hasNext(); System.out.println((new StringBuilder()).append(" - ").append(value.toString()).toString()))
  value = (DataType)i$.next();

System.out.println((new StringBuilder()).append("Selecting 
").append(argValues.get(selection)).append(""").toString());
prevArgValues.add(argValues.get(selection));
}
```

```java
private static final String OES_FUNCTION = "validateAddress";
private static final InspectableFunction OES_FUNCTION_CLASS = new Address();
private static final int ADDRESS_COUNTRY = 0;
private static final int ADDRESS_CITY = 1;
private static final int ADDRESS_STREET = 2;
private static final int ADDRESS_HOUSE = 3;
private static final int ADDRESS_MAILBOX = 4;
private static final int NUM_ARGUMENTS = 5;
private static final DataType CITY_NYC = new OpssString("New York");
private static final DataType CITY_MONTREAL = new OpssString("Montreal");
private static final DataType STREET_VANNESS = new OpssString("Van Ness Str");
private static final DataType STREET_SHERBROOKE = new OpssString("Sherbrooke");
private static final DataType HOUSE_20 = new OpssInteger(20);
private static final DataType HOUSE_10 = new OpssInteger(10);
private static final DataType MAILBOX_FALSE = new OpssBoolean(false);
```

### Example 7–8  Sample Code for getArgValues() Method

```java
public ArrayList<oracle.security.jps.service.policystore.info.DataType>
    getArgValues (String functionName, int argNumber,
    ArrayList<oracle.security.jps.service.policystore.info.DataType> prevArgValues)
    throws RuntimeException {

    ArrayList<oracle.security.jps.service.policystore.info.DataType> values =
        new ArrayList<oracle.security.jps.service.policystore.info.DataType>();

    if (functionName.compareToIgnoreCase(Constants.GET_STRING_IDC_FUNC_NAME) == 0 ||
    functionName.compareToIgnoreCase(Constants.GET_INTEGER_IDC_FUNC_NAME) == 0 ||
    functionName.compareToIgnoreCase(Constants.GET_BOOLEAN_IDC_FUNC_NAME) == 0) {
        if (argNumber == Constants.GET_IDC_FUNC_ARGS.CLAIM_NAME.ordinal()) {
            // return claims in the dictionary
            Iterator<ClaimSchema> it = dictCtx.getDictionary().getClaimsForAllNamespaces();
            while (it.hasNext()) {
                if ((functionName.compareToIgnoreCase(Constants.GET_STRING_IDC_FUNC_NAME) == 0 &&
                it.next().getType() == String.class) ||
                functionName.compareToIgnoreCase(Constants.GET_INTEGER_IDC_FUNC_NAME) == 0 &&
                it.next().getType() == Integer.class) ||
                functionName.compareToIgnoreCase(Constants.GET_BOOLEAN_IDC_FUNC_NAME) == 0 &&
                it.next().getType() == Boolean.class) {
                    values.add (it.next().getUniqueName());
                }
            }

```
Collections.sort(nameList);
for (String name : nameList)
    values.add (new OpssString (name));

} else {
  throw new RuntimeException ("Invalid argument number "+ argNumber);
}

} else {
  throw new RuntimeException ("Invalid function name "+ functionName);
}

return values;

Example 7–9  Sample Code For isValidArgValue() Method

class Example {
  public boolean isValidArgValue (String functionName, int argNumber,
                                 Class argValue, String prevArgValues) 
  
      throws RuntimeException {

          if (functionName.compareToIgnoreCase(Constants.GET_STRING_IDC_FUNC_NAME) == 0 ||
             functionName.compareToIgnoreCase(Constants.GET_INTEGER_IDC_FUNC_NAME) == 0 ||
             functionName.compareToIgnoreCase(Constants.GET_BOOLEAN_IDC_FUNC_NAME) == 0) {
            if (argNumber == Constants.GET_IDC_FUNC_ARGS.CLAIM_NAME.ordinal())
            {
                // is it a valid claim?
                try {
                    ClaimSchema schema = dictCtx.getDictionary().getClaimSchema(argValue.toString());
                    return ((functionName.compareToIgnoreCase(Constants.GET_STRING_IDC_FUNC_NAME)
                    == 0 && schema.getType() == String.class) ||
                    (functionName.compareToIgnoreCase(Constants.GET_INTEGER_IDC_FUNC_NAME)
                    == 0 && schema.getType() == Integer.class) ||
                    (functionName.compareToIgnoreCase(Constants.GET_BOOLEAN_IDC_FUNC_NAME)
                    == 0 && schema.getType() == Boolean.class));
                } catch (UnknownClaimException e) {
                    return false;
                }
            } else {
                throw new RuntimeException ("Invalid argument number "+ argNumber);
            }
        } else {
            throw new RuntimeException ("Invalid function name "+ functionName);
        }
    }
}
Chapter 8

Using the JSP Standard Tag Library

The JavaServer Pages Standard Tag Library (JSTL) consists of custom JavaServer Pages (JSP) elements that encapsulate recurring tasks. Custom tags are reusable JSP components that contain the objects to implement the tasks. They are distributed in a tag library. Oracle Entitlements Server contains custom tags that will call the authorization API. Developers can use these tags in JSP to build a security-based web application. The sections in this chapter contain information on the custom Oracle Entitlements Server JSP tags.

- Section 8.1, "Using the Tag Library"
- Section 8.2, "Defining the Functional Tags"
- Section 8.3, "Defining the Assistant Tags"

8.1 Using the Tag Library

When using the JSTL, you must define the directives in your JSP as follows:

```jsp
<%@ taglib uri="http://www.oracle.com/oes/utils/tags" prefix="oes" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Additionally, add the oestags.jar to the classpath. oestags.jar is located in the OES_CLIENT_HOME/oessm/oestags/ directory in a Security Module installation or in the OES_ADMIN_HOME/oes/oestags/ directory in an Oracle Entitlements Server Administration Server installation.

8.2 Defining the Functional Tags

These functional JSP tags capture the authorization features on Oracle Entitlements Server. The following sections contain information on these functional tags.

- isAccessAllowed Tag
- isAccessNotAllowed Tag
- getUserRoles Tag
- isUserInRole Tag

8.2.1 isAccessAllowed Tag

isAccessAllowed checks if the user is authorized to access a specific resource. If access is allowed, display the body of the tag; if not, skip the body. This is a cooperative and a conditional tag. It will return true or false, and a variable to the body of the JSP which can be used to process obligations.
Defining the Functional Tags

Table 8–1 documents the isAccessAllowed tag definition.

<table>
<thead>
<tr>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>Description: The resource used when calling isAccessAllowed.</td>
</tr>
<tr>
<td></td>
<td>Mandatory</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
<tr>
<td>resourceType</td>
<td>Description: The type of resource used when calling isAccessAllowed.</td>
</tr>
<tr>
<td></td>
<td>If not set, the global resource type set by setSecurityContext will be</td>
</tr>
<tr>
<td></td>
<td>used.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
<tr>
<td>action</td>
<td>Description: The action used when calling isAccessAllowed.</td>
</tr>
<tr>
<td></td>
<td>The default action is view.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
<tr>
<td>resultVar</td>
<td>Description: The name of the scripting variable used to tell if access</td>
</tr>
<tr>
<td></td>
<td>is allowed.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Return Type: boolean</td>
</tr>
<tr>
<td>resultVarScope</td>
<td>Description: The scope of the resultVar (page, request, session, or</td>
</tr>
<tr>
<td></td>
<td>application). The default scope is page.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
<tr>
<td>obligationVar</td>
<td>Description: The name of the variable used for returning obligations</td>
</tr>
<tr>
<td></td>
<td>from the isAccessAllowed call.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Return Type: A map of obligations; the key is the obligation name and</td>
</tr>
<tr>
<td></td>
<td>the value is a map of attributes with attribute names and values.</td>
</tr>
<tr>
<td>obligationVarScope</td>
<td>Description: The scope of the variable containing obligations from</td>
</tr>
<tr>
<td></td>
<td>isAccessAllowed (page, request, session, or application). The default</td>
</tr>
<tr>
<td></td>
<td>scope is page.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
</tbody>
</table>

Note: If you want to show JSP content by tag body, the then/else tag must be used. JSP content cannot be written in the tag body directly without then/else.

Example 8–1 illustrates how isAccessAllowed may be used.

Example 8–1 isAccessAllowed Tag Example

```jsp
<%-- Set global attributes --%>
<oes:setSecurityContext appid="TagLibraryApp" resourceType="image"
    resourcePrefix="images/">
<oes:attribute name="test_attr" value="good_job"/>
</oes:setSecurityContext>
```
Defining the Functional Tags

Using the JSP Standard Tag Library

8.2.2 isAccessNotAllowed Tag

isAccessNotAllowed checks if the user is not authorized to access a specific resource. If access is not allowed, display the body of the tag; if it is, skip the body. This is a cooperative and a conditional tag. It will return true or false, and a variable to the body of the JSP that can be used later to process obligations.

Note: If you want to show JSP content by tag body, the then/else tag must be used. JSP content cannot be written in the tag body directly without using then/else.

Table 8–2 documents the isAccessNotAllowed tag definition.
Defining the Functional Tags

Example 8–2 illustrates how `isAccessNotAllowed` may be used.

**Example 8–2 ** `isAccessNotAllowed` Tag Example

```jsp
<%-- Test for isAccessNotAllowed tag --%>
<oes:isAccessNotAllowed resource="<%=resourceStr %>">
  action="<%=actionStr %>">
  resultVar="isNotAllowed"
  obligationVar="obligations_not"
</oes:isAccessNotAllowed>
<oes:then>
  You have not the permission to <%=actionStr %> the image <%=resourceStr %>.  
  <br/>
  The obligations are: <br/>
  <c:forEach items="${obligations_not}" var="entry">
  <c:out value="${entry.key}" /> &nbsp;=&nbsp; 
  <c:out value="${entry.value}" /> <br/> 
  </c:forEach>
</oes:then>
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td><strong>Description:</strong> The resource used when calling <code>isAccessAllowed</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Mandatory</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
<tr>
<td>resourceType</td>
<td><strong>Description:</strong> The type of resource used when calling <code>isAccessAllowed</code>. If not set, the global resource type set by <code>setSecurityContext</code> will be used.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
<tr>
<td>action</td>
<td><strong>Description:</strong> The action used when calling <code>isAccessAllowed</code>. The default action is view.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
<tr>
<td>resultVar</td>
<td><strong>Description:</strong> The name of the scripting variable used to tell if access is allowed.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> boolean</td>
</tr>
<tr>
<td>resultVarScope</td>
<td><strong>Description:</strong> The scope of the resultVar (page, request, session, or application). The default scope is page.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
<tr>
<td>obligationVar</td>
<td><strong>Description:</strong> The name of the variable used for returning obligations from the <code>isAccessAllowed</code> call.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> A map of obligations. The key is the obligation name and the value is a map of attributes with attribute names and values.</td>
</tr>
<tr>
<td>obligationVarScope</td>
<td><strong>Description:</strong> The scope of the variable containing obligations from <code>isAccessAllowed</code> (page, request, session, or application). The default scope is page.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
</tbody>
</table>
<oes:else>
You have the permission to <%=actionStr %> the image <%=resourceStr %>.<br/>
<img src="images/private.jpg" width="250" height="150" />
The obligations are:<br/>
<c:forEach items="${obligations_not}" var="entry">
<c:out value="${entry.key}" /> &nbsp;=&nbsp; <c:out value="${entry.value}" /> <br/>
</c:forEach>
</oes:else>
</oes:isAccessNotAllowed>

<%-- another way to use tag isAccessNotAllowed --%>
<oes:isAccessNotAllowed resource="<%=resourceStr %>">
  action="<%=actionStr %>"> resultVar="isNotAllowed" 
  obligationVar="obligations_not" />
  <c:choose>
  <c:when test="${isNotAllowed}">You have not the permission to <br/>
  <%=actionStr %> the image <%=resourceStr %>.<br/>
  </c:when>
  <c:otherwise>
  You have the permission to <%=actionStr %> the image <%=resourceStr %>.<br/>
  <img src="images/private.jpg" width="250" height="150" />
  The obligations are:<br/>
  <c:forEach items="${obligations}" var="entry">
  <c:out value="${entry.key}" /> &nbsp;=&nbsp; <c:out value="${entry.value}" /> <br/>
  </c:forEach>
  </c:otherwise>
  </c:choose>

8.2.3 getUserRoles Tag

getUserRoles retrieves the roles assigned to the user for a particular resource and action. This is a cooperative tag that returns a variable to the JSP that can be used later for processing. Table 8–3 documents the getUserRoles tag definition.

<table>
<thead>
<tr>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>Description: The resource used when calling getUserRoles. Mandatory</td>
</tr>
<tr>
<td>resourceType</td>
<td>Description: The type of resource used when calling getUserRoles; If it is not set, the global resource type set by setSecurityContext will be used. Optional Return Type: not applicable</td>
</tr>
<tr>
<td>action</td>
<td>Description: The action used when calling getUserRoles. The default action is view. Optional Return Type: not applicable</td>
</tr>
</tbody>
</table>
Defining the Functional Tags

Example 8–3 illustrates how `getUserRoles` may be used.

**Example 8–3  `getUserRoles` Tag Example**

```xml
<%-- Test for tag getUserRoles --%>
<oes:setSecurityContext appId="TagLibraryApp" resourceType="jspfile"
resourcePrefix="">
  <oes:attribute name="myroleattr" value="its_my_role"/>
</oes:setSecurityContext>
<oes:getUserRoles resource="protected/rolepolicy.jsp" action="write"
resultVar="rolenames" />
<c:out value="Role names are:
"/>
<c:forEach items="${rolenames}" var="rolename">
  <c:out value="${rolename}" /> <br>
</c:forEach>
```

8.2.4 `isUserInRole` Tag

`isUserInRole` checks if the user has been assigned to the specified role for a particular resource and action. This is a cooperative and a conditional tag. It will return true (if the current user has a specific role) or false, and a result variable to the body of the JSP for later processing.

<table>
<thead>
<tr>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>role</td>
<td><strong>Description:</strong> The name of the role to check against the user.</td>
</tr>
<tr>
<td></td>
<td><strong>Mandatory</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
<tr>
<td>resource</td>
<td><strong>Description:</strong> The name of the resource against which to check the user's roles.</td>
</tr>
<tr>
<td></td>
<td><strong>Mandatory</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
</tbody>
</table>

**Note:** If you want to show JSP content by tag body, the `then/else` tag must be used. JSP content cannot be written in the tag body directly without using `then/else`.

Table 8–4 documents the `isUserInRole` tag definition.
Example 8–4 illustrates how `isUserInRole` may be used.

**Example 8–4  `isUserInRole` Tag Example**

```jsp
<%@-- Test for tag `isUserInRole` --%>
<oes:isUserInRole role="tagrole1" resource="protected/rolepolicy.jsp" action="write" resultVar="isUserInRole" resultVarScope="request">
  <oes:then>You are in the role "tagrole1".</oes:then>
  <oes:else>You are not in the role "tagrole1".</oes:else>
</oes:isUserInRole>

<%@-- we can also use following scripts to test if the user is in the specific role --%>
<c:choose>
  <c:when test="${isUserInRole}">
    <iframe src="protected/rolepolicy.jsp?isUserInRole=<c:out value='${isUserInRole}'/>" width="500" height="250" />
  </c:when>
  <c:otherwise>
    You are not in role "tagrole1", and cannot see the content of protected/rolepolicy.jsp
  </c:otherwise>
</c:choose
```

**Table 8–4 (Cont.) `isUserInRole` Tag Definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>resourceType</td>
<td>Description: The type of resource against which to check the user's roles. If it is not set, the global resource type set by <code>setSecurityContext</code> will be used.</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
<tr>
<td>action</td>
<td>Description: The resource's action against which the user's role will be checked. The default value will be view.</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
<tr>
<td>resultVar</td>
<td>Description: A variable used to hold the result from <code>isUserInRole</code> for later use.</td>
</tr>
<tr>
<td></td>
<td>Return Type: boolean</td>
</tr>
<tr>
<td>resultVarScope</td>
<td>Description: The scope of the <code>resultVar</code> (page, request, session, or application). The default scope is page.</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
</tbody>
</table>

Example 8–4 illustrates how `isUserInRole` may be used.

8.3 Defining the Assistant Tags

Assistant (also known as non-functional) tags are helper tags. The following sections contain information on these assistant tags.

- `setSecurityContext` Tag
- `attribute` Tag
- `then/else` Tags
8.3.1 setSecurityContext Tag

setSecurityContext is a cooperative tag that will set up data (including the application ID, Resource Type and the prefix of the resource name for other tags) in the specified page scope. The attributes that should be set globally in the application context can be set in the body of this tag using the attribute tag (as described in Section 8.3.2, “attribute Tag”). The attributes set by setSecurityContext will then be put into the application context as its authorization call elements. Table 8–5 documents the setSecurityContext tag definition.

<table>
<thead>
<tr>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>appId</td>
<td><strong>Description:</strong> The appId of the security context that will be used to construct the runtime resource for all other tags on the page that have a resource attribute.</td>
</tr>
<tr>
<td></td>
<td><strong>Mandatory</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
<tr>
<td>resourceType</td>
<td><strong>Description:</strong> The global resource type which can be used by all other authorization tags.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
<tr>
<td>resourcePrefix</td>
<td><strong>Description:</strong> The prefix of the resource name. If most of the resources on one JSP have the same prefix, this attribute can be used to shorten the resource name for each authorization tag. For example, if there are many images protected by the Authorization Policy under /product/cat1/images/, the prefix can set as /product/cat1/images/ and the resource name would be the simple image name such as mobile.jpg.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
</tbody>
</table>

Example 8–5 illustrates how setSecurityContext may be used.

**Example 8–5  setSecurityContext Tag Example**

```jsp
<%-- Set global attributes --%>
<oes:setSecurityContext appId="TagLibraryApp" resourceType="image"
resourcePrefix="/images/">
<oes:attribute name="test_attr" value="good_job"/>
</oes:setSecurityContext>
```

8.3.2 attribute Tag

attribute is a tag that can be used to pass extra variables into the Oracle Entitlements Server application context by other Oracle Entitlements Server JSP tags. These variables will be used to write constraints against Authorization Policies. Table 8–6 documents the attribute tag definition.

<table>
<thead>
<tr>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td><strong>Description:</strong> The name of the attribute to set in the application context.</td>
</tr>
<tr>
<td></td>
<td><strong>Mandatory</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> not applicable</td>
</tr>
</tbody>
</table>
### Example 8–6 illustrates how `attribute` may be used.

#### Example 8–6 `attribute` Tag Example

```xml
<oes:attribute name="myroleattr" value="its_my_role"/>
```

### 8.3.3 `then/else` Tags

`then/else` is a tag used for displaying content for conditional tags (including `isAccessAllowed`, `isAccessNotAllowed` and `isUserInRole`). If the result of the conditional tags is true, the content in the tag `then` is displayed; otherwise the content in the tag `else` is displayed. These tags are simple tags with no additional definition.

<table>
<thead>
<tr>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Description: The value of the attribute to set in the application context.</td>
</tr>
<tr>
<td></td>
<td>Mandatory</td>
</tr>
<tr>
<td></td>
<td>Return Type: not applicable</td>
</tr>
</tbody>
</table>
Index

A

actions
  adding, 2-11
  policy, 1-3
adding fine grained components, 1-4
administration
  delegating, 5-1
administration roles, 5-1
  ApplicationPolicyAdmin, 5-3
  assigning principals, 5-5
  creating, 5-3, 5-4
  default, 5-3
  definition, 5-2
  manage, 5-3
  managing, 5-6
PolicyDomainAdmin, 5-3
  system, 5-2
  SystemAdmin, 5-3
  view, 5-3
administrator
  application policy, 5-2
  policy domain, 5-2
AdminManager, 5-4
AdminResourceActionEntry, 5-4
adminRole, 5-4
AdminRoleEntry
  managing, 5-6
advanced policy, 1-4
advanced policy elements, 2-16
application
  see ApplicationPolicy, 2-7
ApplicationPolicy
  application policy, 5-2
  administrator, 5-2
  creating, 2-7
application role object, 2-16
application roles
  creating, 1-4
  hierarchy, 2-17
  managing, 3-5
ApplicationPolicy
  bind to Security Module, 4-4
  creating, 2-7
  managing, 3-3
  scope level, 3-2
ApplicationPolicyAdmin administration role, 5-3
AppRoleEntry, 1-4, 2-16
  managing, 3-5
AppRoleManager, 1-4
attribute retrievers, 7-1
  and jps-config.xml, 7-1
  custom, 7-2
attribute tag, 8-8
AttributeEntry, 1-6, 2-19
  managing, 3-7
AttributeRetrieverV2 interface, 7-2
  implementing, 7-3
attributes
  as extensions, 2-19
authorization policy
  and role mapping policy, 2-17, 3-6

B

BasicPolicyRuleEntry, 2-13
binding
  Security Module, 4-4
  SMEEntry, 4-6
boolean
  resourceActionOnly, 6-11
boolean expressions
  constraint, 2-25
BooleanExpressionEntry, 2-22

cascadeDelete, 3-5
checkPermission() calls, 6-21
comparison operators, 2-3
complex search, 2-2
constraint, 2-22
  adding, 1-6
  boolean expressions, 2-25
  function expressions, 2-26
controlled distribution, 4-3
controlled pull distribution, 4-3
controlled push distribution, 4-3
create method
  overview, 2-2
custom attribute retrievers, 7-2
custom functions, 7-6

data types, 2-3
default administration roles, 5-3
delegated administration
  overview, 5-1
  scope, 5-2
delete method
  overview, 2-2
deleteRolePolicy(), 3-6
distribution modes, 4-3
  controlled, 4-3
  non-controlled, 4-4
dynamic attribute, 2-20

entitlement, 1-6
Extension, 1-6
ExtensionManager, 1-6
extensions
  attributes, 2-19
  functions, 2-19
  managing, 3-7, 3-8

fine grained elements, 2-16
fine grained policy, 1-4
function expressions
  constraint, 2-26
FunctionEntry, 1-6, 2-20
  managing, 3-8
functions
  as extensions, 2-19

gateway
  XACML, 6-22
getGrantedAdminResources, 5-5
getRolePolicy(), 3-6
getSecurityContext tag, 8-8
getUserRoles tag, 8-5
grantAdminRole, 5-5
granularity
  delegated administration, 5-2

hierarchical resources, 2-9, 2-11
hierarchy
  application roles, 2-17

InspectableFunction, 7-6, 7-9
isAccessAllowed tag, 8-1
isAccessNotAllowed tag, 8-3

Java API
  create method, 2-2
  delete method, 2-2
  manager interfaces, 2-2
  modify method, 2-2
  policy objects, 2-1
  search query, 2-2
jps-config.xml, 2-7
  and attribute retrievers, 7-1
JSP tags
  see tags, 8-1

manage privileges, 5-3
management
  scoping, 3-1
  manager interfaces, 2-2
  managing
    SMEntry, 4-5
    modify method
      overview, 2-2
      modifyRolePolicy(), 3-6

non-controlled distribution, 4-4

object
  PermissionSetEntry, 1-6
objects
  AdminResourceActionEntry, 5-4
  adminRole
    creating, 5-4
  AdminRoleEntry, 5-6
  ApplicationPolicy, 1-3, 2-7
    managing, 3-3
  AppRoleEntry, 1-4, 2-16
    managing, 3-5
  AttributeEntry, 1-6, 2-19
    managing, 3-7
  FunctionEntry, 1-6, 2-20
    managing, 3-8
    managing
      PolicyStore, 3-3
  ObligationEntry, 1-6, 2-27
  PermissionSetEntry, 2-21
    managing, 3-10
  PolicyDomainEntry, 1-3
    creating, 5-6
    managing, 3-4, 3-12
  PolicyEntry, 1-4, 2-14
    managing, 3-11
  PolicyRuleEntry, 2-13
  PolicyStore, 1-2, 2-6, 3-1
PrincipalEntry, 2-13
ResourceActionsEntry, 1-3, 2-11
ResourceEntry, 1-3, 2-9
  managing, 3-9
ResourceTypeEntry, 1-3, 2-8
  managing, 3-5
RolePolicyEntry, 3-6
RuleExpressionEntry, 2-22
SMEntry, 4-4
  binding, 4-6
  managing, 4-5
obligation
  building, 1-6
obligation object, 2-27
ObligationEntry, 1-6, 2-27
obligations, 2-27

P

PEP API
  calls, 6-1
Permission Set
  managing, 3-10
permission set
  populating, 1-6
Permission Set object, 2-21
PermissionSetEntry, 1-6, 2-21
  managing, 3-10
PIP
  and attribute retrievers, 7-1
policy
  actions, 1-3
  adding advanced elements, 2-16
  adding fine grained elements, 1-4
  and roles, 1-7
  building, 1-4
  components, 1-1
  composing simple, 1-2
  consolidating, 2-14
  constraint, 1-6
  executing simple, 2-6
  managing, 3-11, 3-12
  obligation, 1-6
policy distribution
  initiating, 4-7
  overview, 4-1
policy domain
  administrator, 5-2
  creating, 5-6
  default, 1-3
  managing, 3-4
policy objects
  and API, 2-1
policy rule, 2-13
policy simple components, 1-2
policy store, 1-2
  accessing, 2-6
  defining, 3-1
PolicyDomainAdmin administration role, 5-3
PolicyDomainEntry, 1-3
  creating, 5-6
  managing, 3-4, 3-12
  scope levels, 3-2
PolicyEntry, 1-4
  consolidating, 2-14
  managing, 3-11
PolicyManager, 3-11
PolicyRuleEntry, 1-4, 2-13
PolicyStore, 1-2
  accessing, 2-6
  defining, 3-1
  managing objects, 3-3
principal, 2-13
PrincipalEntry, 1-4, 2-13
principals
  assigning, 5-5
  retrieving resources, 5-5
privileges
  assigning, 5-4
  manage, 5-3
  view, 5-3
pull (policy distribution), 4-3
push (policy distribution), 4-3

R

RBAC
  and delegating administration, 5-1
resource
  instantiating, 2-9
  managing, 3-9
resource attribute, 2-20
resource attributes, 2-8
resource object, 1-3
resource type
  creating, 2-8
  managing, 3-5
resource type object, 1-3
resourceActionOnly, 6-11
ResourceActionsEntry, 1-3, 1-4
  creating, 2-11
ResourceEntry, 1-3
  hierarchical, 2-9, 2-11
  instantiating, 2-9
  managing, 3-9
ResourceIdManager, 1-3
ResourceTypeEntry, 1-3
  creating, 2-8
  hierarchical, 2-9, 2-11
  managing, 3-5
ResourceTypeManager, 1-3
role catalog, 2-16, 3-5
role category, 3-5
role mapping policy, 2-16, 3-6
  and authorization policy, 2-17, 3-6
  managing, 3-6
  overview, 1-5
roles
  implementing policy, 1-7
RuleExpressionEntry, 2-22
S
scope
  delegated administration, 5-2
scope levels, 3-1
  ApplicationPolicy, 3-2
  PolicyDomainEntry, 3-2
search query
  overview, 2-2
  simple and complex, 2-2
Security Module
  bind to ApplicationPolicy, 4-4
  simple policy, 1-2, 2-6
  simple search, 2-2
  SMEntry
    binding, 4-6
    managing, 4-5
  system administrator, 5-2
  SystemAdmin administration role, 5-3

T
tag, 8-1
  attribute, 8-8
  getSecurityContext, 8-8
  getUserRoles, 8-5
  isAccessAllowed, 8-1
  isAccessNotAllowed, 8-3
  isUserInRole, 8-6
  then/else, 8-9
  then/else tag, 8-9

U
use cases
  attribute retrievers, 7-1

V
view privileges, 5-3

X
XACML gateway, 6-22