Oracle® Fusion Middleware

Developer's Guide for Oracle Entitlements Server

11g Release 1 (11.1.1)

E14097-03

August 2011

ORACLE®

Oracle Fusion Middleware Developer's Guide for Oracle Entitlements Server 11g Release 1 (11.1.1)

E14097-03

Primary Author: Michael Teger

Contributing Author:

Contributor:

Contents

3 Managing Policy Objects Programmatically

4 Distributing Policies

5 Delegating Policy Administration

List of Examples

List of Figures

List of Tables

Preface

The *Oracle Fusion Middleware Developer's Guide for Oracle Entitlements Server* 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 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

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. 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`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers 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:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Using the Policy Model

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:

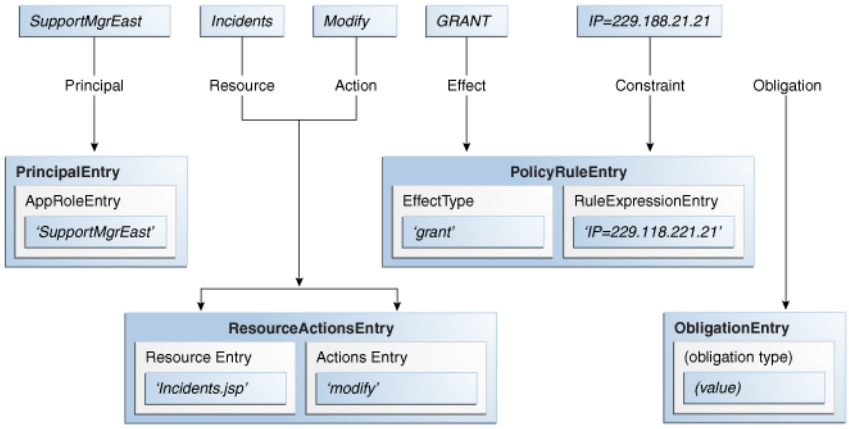
1.1 Examining Policy Elements

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 (also discussed in the *Oracle Fusion Middleware Administrator's Guide for Oracle Entitlements Server*):

```
GRANT the SupportManagerEast role MODIFY access to the Incidents servlet
  if the request is made from an IP address of 229.188.21.21
```

Figure 1–1 illustrates how these elements map to policy-related objects (in the policy model) that can be used to create policies programmatically.

Figure 1–1 Policy Components Mapped to Policy Objects



This image illustrates how general policy components map to Oracle Entitlements Server policy objects.

****

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 2.2.1, "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 are used to define rules about how principals are granted or denied roles. See Chapter 7, "Extending Functionality" for more information.

1.2 Composing A Simple Policy

Composing a simple 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 2.2.1, "Accessing the Policy Store."

Note: A policy store is created and configured during installation of Oracle Entitlements Server. For information, see *Oracle Fusion Middleware Installation Guide for Oracle Identity Management*.

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 2.2.2, "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 2.2.3, "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 2.2.4, "Instantiating a Resource."

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 for a `ResourceEntry` object are a subset of the set of legal actions already defined in the `ResourceTypeEntry` it references. A `ResourceActionsEntry` object can be added directly to a policy or one or more can be added to a `PermissionSetEntry`. For general information, see Section 1.3.4, "Populating a Permission Set." For programming details on adding `ResourceActionsEntry` objects to a `PermissionSetEntry` object, see Section 2.3.4, "Defining Permission Sets."

6. Build the `PolicyEntry`.

   This includes:

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

      See Section 2.2.6, "Specifying a Policy Rule" for more information.

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

      See Section 2.2.7, "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 2.2.5, "Associating Actions with the Resource" for more information.

   d. Calling the `PolicyManager` and creating the `PolicyEntry`.

      See Section 2.2.8, "Defining the Policy" for more information.

7. Distribute the policy.

   See Chapter 4, "Distributing Policies" 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 role 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 a subject assigned to a role (using a Role Mapping Policy) also inherits any child roles as long as it is not prohibited by other Role Mapping Policies. When an `AppRoleEntry` is referenced as a policy principal, access to the resource of all users assigned the role is governed by the policy. For more information, see Section 2.3.1, "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 2.3.2, "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 effect defined in the `PolicyRuleEntry` is applicable.
- If the expression does not resolve to true, the policy is not applicable.

A Condition must be true for the `PolicyRuleEntry` to evaluate to a positive effect. 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, call the `ExtensionManager` and create an object to be referenced in the policy. The `ExtensionManager` provides methods to create and manage an `AttributeEntry` (a name/value pair that can be dynamically added to a policy rule) or a `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 2.3.5, "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 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 2.3.4, "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 2.3.6, "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), a 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 2.3.1, "Creating Application Roles" and Section 2.3.2, "Creating Role Mapping Policies."

2

Constructing A Policy Programmatically

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

- Section 2.1, "Using the Java API"
- Section 2.2, "Executing A Simple Policy"
- Section 2.3, "Creating Fine Grained Elements for a Simple Policy"

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.
2. Retrieve the application policy under which the object will be (or has been) created.

   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 list documents more specifically how the API can be used to perform specific operations on policy objects.

- 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 2.2, "Executing A Simple Policy" and Section 2.3, "Creating Fine Grained Elements for a Simple Policy" for examples of the `create` method and descriptions of its parameters.

- 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 propogate 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.
- 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.
- 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 mutiple 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

Parameter	Description
policy_object`.SEARCH_PROPERTY`	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.
negation	A boolean that takes as a value either `true` or `false`. If true, the NOT operator is applied to the search.
operator	An enum that defines the `ComparatorType` as one of the following: ▪ EQUALITY ▪ GREATER THAN ▪ GREATER THAN OR EQUAL TO ▪ LESS THAN ▪ LESS THAN OR EQUAL TO
search string	Takes as a value the string used for the search. ▪ If the value is null, the match must be ANY. ▪ If populated, the algorithm matches the value against those being searched.

Table 2–1 (Cont.) Using the Complex SearchQuery Parameters

Parameter	Description
`SearchQuery.MATCHER`	An enum that defines how the search string is matched against the values being searched. It should define one of the following: ■ 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. ■ BEGINS_WITH — The object property must begin with the search string. ■ CONTAINED_IN — The object property must contain the search string. ■ ENDS_WITH — The object property must end with the search string. ■ EXACT — The object property must be exactly the same as the search string.

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.

- The following data types are supported by Oracle Entitlements Server.
  - String
  - Time
  - Date
  - Integer

  Table 2–2 lists the comparison operator options organized by data type.

Table 2–2 Available Comparison Operators by Data Type

Expression Data Type	Available Comparison Operator
String	STRING_EQUAL (supported when comparing multi-value attributes only)
	STRING_REGEXP_MATCH
	STRING_IS_IN
Boolean	BOOLEAN_EQUAL (supported when comparing multi-value attributes only)
Date	DATE_EQUAL (supported when comparing multi-value attributes only)
	DATE_GREATER_THAN
	DATE_GREATER_THAN_OR_EQUAL
	DATE_LESS_THAN
	DATE_LESS_THAN_OR_EQUAL
	VALID_UNTIL_DATE
	DATE_IS_IN

Table 2–2 (Cont.) Available Comparison Operators by Data Type

Expression Data Type	Available Comparison Operator
Integer	INTEGER_EQUAL (supported when comparing multi-value attributes only)
	INTEGER_GREATER_THAN
	INTEGER_GREATER_THAN_OR_EQUAL
	INTEGER_LESS_THAN
	INTEGER_LESS_THAN_OR_EQUAL
	INTEGER_IS_IN
	NOT
Time	TIME_EQUAL (supported when comparing multi-value attributes only)
	TIME_GREATER_THAN
	TIME_GREATER_THAN_OR_EQUAL
	TIME_LESS_THAN
	TIME_LESS_THAN_OR_EQUAL
	VALID_FOR_HOURS
	VALID_FOR_MINUTES
	VALID_FOR_MSECONDS
	VALID_FOR_SECONDS
	VALID_UNTIL_TIME
	TIME_IS_IN

For detailed information on the Oracle Entitlements Server Java API, see one or both of the 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*

2.2 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 2.2.1, "Accessing the Policy Store"
- Section 2.2.2, "Creating an Application Policy"
- Section 2.2.3, "Defining Resource Types"
- Section 2.2.4, "Instantiating a Resource"
- Section 2.2.5, "Associating Actions with the Resource"
- Section 2.2.6, "Specifying a Policy Rule"
- Section 2.2.7, "Specifying the Principal"

- Section 2.2.8, "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."

2.2.1 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`.

```
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 (OPSS), `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. See the *Oracle Fusion Middleware Security Guide* for more information on this configuration file. Parameters specific to Oracle Entitlements Server are documented in the *Oracle Fusion Middleware Administrator's Guide for Oracle Entitlements Server*.

> **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.

```
Subject smith = new Subject();
Principal principal = new WLSUserImpl("smith");
```

```
smith.getPrincipals().add(principal);
BindingPolicyStore ps = BindingPolicyStoreFactory.getInstance();
ps.setSubject(smith);
```

It is assumed the user is already authenticated.

2.2.2 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."

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

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.

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

2.2.3 Defining Resource Types

A `ResourceTypeEntry` object specifies the full scope of traits for a particular kind of protected resource. It contains resource attributes and definitions of all possible valid actions that can be performed on the protected 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` must include the full range of actions that may be granted or denied on any `Resource` instance of this type. 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 = Trading.getResourceTypeManager();
List<String> actions = new ArrayList();
   actions.add("get");
List<AttributeEntry<? extends DataType>> attributes = new
ArrayList<AttributeEntry<? extends DataType>>();
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 2.3.3, "Creating Attribute and Function Definitions" for more information. Attributes can also be null.
- Delimiter - `/` (forward slash) is the default delimeter 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 list of actions supplied under Actions are validated at runtime by the implementation.

`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. Additional information is noted in Section 2.2.4, "Instantiating a Resource."

2.2.4 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). See the *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.

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.

> **Note:** A `ResourceEntry` object is defined as an instance of a `ResourceType` object. Be sure the appropriate `ResourceType` is defined before attempting to create a `ResourceEntry` instance. For more information, see Section 2.2.3, "Defining Resource Types."

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

```
ResourceManager resMgr = Trading.getResourceManager();
ResourceEntry checkingRes = resMgr.createResource("Bob_checking1",
 "Bob Checking Account", "Checking account.", resType, null);
```

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 - null specifies that there are no (optional) attributes being configured for this `ResourceEntry`. To define one or more attributes, create them with the `createAttribute()` method in the `ExtensionManager` and reference them here (instead of null).

Once a ResourceEntry is created, it can be paired with actions in a ResourceActionsEntry or included in a PermissionSetEntry. For more information, see Section 2.3.4, "Defining Permission Sets."

Note: As noted in Section 2.2.3, "Defining Resource Types," hierarchical ResourceEntry objects can be instantiated from ResourceTypeEntry objects. When instantiating a hierarchical ResourceEntry object:

- The name of the ResourceEntry must start with a delimiter. For example, if the delimiter is /, a valid name is /region/East.
- All parent resources must already be created. For example, a resource /region/East/NY can only be created if both /region and /region/East have already been created.

2.2.5 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 2.2.5.1, "Using a ResourceEntry"
- Section 2.2.5.2, "Using a ResourceNameExpression"

2.2.5.1 Using a ResourceEntry

The procedure to instantiate a ResourceEntry is explained in Section 2.2.4, "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>();
resActsList.add(new BasicResourceActionsEntry(Checking, <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.

2.2.5.2 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 2–3 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 2–3 Matching ResourceNameExpression Objects

ResourceEntry	ResourceActionsEntry	Policies
ResType1 (HTTP URL)	RAE1 with ResType1 `ResourceEntry` http://www.oracle.com and action GET	Policy1 with RAE1
ResType2 (HTTPS URL)	RAE2 with ResType2 `ResourceEntry` https://www.oracle.com and action GET	Policy2 with RAE2
	RAE3 with ResType1 `ResourceNameExpression` http://* and action GET	Policy3 with RAE3

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

Example 2–5 Building a ResourceActionsEntry with ResourceNameExpression

```
// 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 2–4 has examples of the `ResourceNameExpression`. While any regular expression can be used, the pattern expressions listed in the table are processed faster then regular expresisons.

Table 2–4 Examples of ResourceNameExpression

Expression	Description
Specific to Resource Type	To specify any action type, use the keyword specific to the Resource Type. `resActsList.add(new BasicResourceActionsEntry(checkingRes,"any"));`
".*"	To specify all resources `resActsList.add(new BasicResourceActionsEntry(".*", actions));`
"http.*"	To specify all resources beginning with http. `resActsList.add(new BasicResourceActionsEntry("http.*", actions));`

Table 2–4 (Cont.) Examples of ResourceNameExpression

Expression	Description
".*html"	To specify all resources ending in html. `ResourceActionsEntry suffixResActions = new BasicResourceActionsEntry(".*html", actions);`

You may also populate a Permission Set with one or more `ResourceActionsEntry` objects. See Section 2.3.4, "Defining Permission Sets" for more information.

2.2.6 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 policies must contain one (and only one) Policy Rule. 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
 ("ReportRule", "Report Policy Rule", "Rule for Reports policy.",
 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 2.3.5, "Defining a Condition" for more information.

2.2.7 Specifying the Principal

A `PrincipalEntry` specifies the users, groups, or roles to which the policy pertains. Table 2–5 illustrates these types and how each can be specified programmatically.

Table 2–5 Specifying a Principal Programmatically

Principal Type	Example

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 2.3.1, "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);`

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);`

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.

2.2.8 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,

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

```
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 an ordered collection (list) of `PermissionSetEntry` objects. See Section 2.3.4, "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 2.3.6, "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.

2.3 Creating Fine Grained Elements for a Simple Policy

Section 2.2, "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 2.3.1, "Creating Application Roles"
- Section 2.3.2, "Creating Role Mapping Policies"
- Section 2.3.3, "Creating Attribute and Function Definitions"
- Section 2.3.4, "Defining Permission Sets"
- Section 2.3.5, "Defining a Condition"
- Section 2.3.6, "Adding Obligations"

2.3.1 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 role. Once an `AppRoleEntry` is defined, the `grantAppRole` method can be used to assign the role to a subject statically or a *Role Mapping Policy* can be created to assign it to subjects dynamically. (An *Authorization Policy* is used to define the role's access rights.)

Note: See Section 2.3.2, "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

```
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

```
//create user named JSMITH PrincipalEntry aUser = new
BasicPrincipalEntry("weblogic.security.principal.WLSUserImpl", "JSMITH");

//Add user to principals list
List<PrincipalEntry> principal = new ArrayList<PrincipalEntry>();
principal.add(aUser);

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

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

```
//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> principal = new ArrayList<PrincipalEntry>();
principal.add(traderManagers);

//add TraderManagers role as principal of AllManagers role
```

```
roleMgr.grantAppRole(allManagers, principal);
```

2.3.2 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.

> **Note:** The `RolePolicyEntry` object is represented in the Oracle Entitlements Server Administration Console as a Role Mapping Policy, organized within the Role Catalog.

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

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

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

List<PrincipalEntry> principals = 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(resource1);

//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 2.2.7, "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.

  Note: Conditions in Role Mapping Policies provide the same functionality as conditions in Authorization Policies.

- Resource Names - resources is a list of `ResourceEntry` objects to associate with the Role Mapping Policy. It is an optional parameter for which you can supply null or an empty list. This parameter also allows scoping the Role Mapping Policy to a particular resource(s).
- Resource Name Expressions - This value can contain a list of resource name expressions to associate with the Role Mapping Policy. It is an optional parameter for which you can supply null (as in this example) or an empty list. This parameter also allows scoping the Role Mapping Policy to a particular resource(s).

2.3.3 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 2.3.5, "Defining a Condition" and Section 2.3.6, "Adding Obligations" contain additional coding information.

Attribute and function definitions are managed at the `ApplicationPolicy` level. Pre-defined definitions can be used (`RuleExpressionEntry.BuiltInAttributes` and `RuleExpressionEntry.BuiltInFunctions`) or you can define new ones to suit your requirements using the `ExtensionManager`. More information can be found in the following sections.

- Section 2.3.3.1, "Creating Attribute Definitions."
- Section 2.3.3.2, "Creating Custom Function Definitions."

2.3.3.1 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

```
//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.

2.3.3.2 Creating Custom Function Definitions

A Custom Function represents some externally implemented logic that is used to generate an output which is then 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

```
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 2.3.5, "Defining a Condition" and Section 7.2, "Developing Custom Functions."

2.3.4 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 2.2.5, "Associating Actions with the Resource" for more information on `ResourceActionsEntry` objects.) With the `PermissionSetEntry`, you can bundle `ResourceActionsEntry` objects as needed. 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(Checking, <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.

2.3.5 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 2.2.6, "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:

  ```
  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

```
BooleanExpressionEntry bexp =
  new BooleanExressionEntry(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, check_balance), 10000)`
   - `customFunc_checkCustomerType(username, "GOLD")`
   - `and(integer_greater_than(integer_add (saving_balance, check_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

```
//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 2.3.5.1, "Constructing a Boolean Expression"
- Section 2.3.5.2, "Constructing a Custom Function Expression"

2.3.5.1 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 stings) are basic expressions. The boolean expression compares the results returned by the basic expressions.

```
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).

2.3.5.2 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.

```
//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.

```
// define the acceptable expression value types
List<Class<? 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.

2.3.6 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`.

   ```
   AttributeAssignment<OpssString>traderRpts = 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.

```
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.

```
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.

3

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, "Using Scope Levels for Management"
- Section 3.2, "Managing Objects Created at the PolicyStore Scope"
- Section 3.3, "Managing Objects Within the ApplicationPolicy Scope"
- Section 3.4, "Managing Objects within the PolicyDomainEntry Scope"

3.1 Using Scope Levels for Management

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 2.2.1, "Accessing the Policy Store." The policy store location, the account and the account password used to access it are defined in the `jps-config.xml` 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

```
<jpsConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://
xmlns.oracle.com/oracleas/schema/jps-config-11_0.xsd">
    <serviceProviders>
        <serviceProvider type="POLICY_STORE" name="policy.db"
class="oracle.security.jps.internal.policystore.OPSSPolicyStoreProvider" />
    </serviceProviders>
    <serviceInstances>
        <serviceInstance name="policystore.db" provider="policy.db">
           <property name="repository.type" value="database"/>
           <property name="jdbc.url"
value="jdbc:oracle:thin:@10.182.219.120:1521:mc"/>
           <property name="jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/>
           <property name="jdbc.user" value="wcai"/>
           <property name="jdbc.password" value="password"/>
           <property name="root.dn" value="cn=farm,cn=JPSContext,cn=jpsroot"/>
          <property name="pd.timer.enabled" value="false"/>
```

```
        </serviceInstance>
    </serviceInstances>
    <jpsContexts default="default">
        <jpsContext name="default">
            <serviceInstanceRef ref="policystore.db"/>

        </jpsContext>
    </jpsContexts>
</jpsConfig>
```

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

- An `ApplicationPolicy` object represents an application being secured by Oracle Entitlements Server. Within an `ApplicationPolicy`, programmatic objects used to define policies (Resource Types, Functions, Attributes, Application Roles and Role Policies) are managed.

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

- An optional `PolicyDomainEntry` object can be created to partition, and serve as a management point for, Resources, Permission Sets 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."

  > **Note:** 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."

3.2 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 2.2.2, "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 `deleteApplicationPolicy()` method.

Example 3–2 Using deleteApplicationPolicy() Method

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

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

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

3.3 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 3.3.1, "Managing PolicyDomainEntry Objects"
- Section 3.3.2, "Managing ResourceTypeEntry Objects"
- Section 3.3.3, "Managing and Granting AppRoleEntry Objects"
- Section 3.3.4, "Managing Role Mapping Policy (RolePolicyEntry) Objects"
- Section 3.3.5, "Managing AttributeEntry and FunctionEntry Objects"
- Section 3.3.6, "Managing ResourceEntry Objects"
- Section 3.3.7, "Managing Permission Sets"
- Section 3.3.8, "Managing the Policy"

3.3.1 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

```
PolicyDomainManager domainMgr = Trading.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

```
PolicyDomainManager domainMgr = Trading.getPolicyDomainManager();

PolicyDomainManager domainMgr = Trading.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

```
PolicyDomainManager domainMgr = Trading.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.

3.3.2 Managing ResourceTypeEntry Objects

Section 2.2.3, "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

```
//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 `ResourceType` and related objects would be removed. If true, the `ResourceType` and all instantiated `ResourceEntry` objects are deleted. If false, and any `ResourceEntry` instances exist, the operation fails and `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.

3.3.3 Managing and Granting AppRoleEntry Objects

Section 2.3.1, "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.

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 grantee (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 3.3.4, "Managing Role Mapping Policy (RolePolicyEntry) Objects" for more information.

Revocation of the `AppRoleEntry` can be done using the `revokeAppRole()` method.

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.

3.3.4 Managing Role Mapping Policy (RolePolicyEntry) Objects

Section 2.3.2, "Creating Role Mapping Policies" documents how to create a `RolePolicyEntry` object. You can also delete, modify and retrieve `RolePolicyEntry` 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

```
//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

```
//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.

3.3.5 Managing AttributeEntry and FunctionEntry Objects

Section 2.3.3, "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 3.3.5.1, "Managing AttributeEntry Objects"
- Section 3.3.5.2, "Managing FunctionEntry Objects"

3.3.5.1 Managing AttributeEntry Objects

Example 3–10 retrieves an `AttributeEntry` object named Phone from the policy store. bankApplication refers to the `ApplicationPolicy` object from which the `ExtensionManager` is instantiated. `Phone` refers to the unique identifier defined as the Name when the `AttributeEntry` object was initially created.

Example 3–10 Using the getAttribute() Method

```
//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.

3.3.5.2 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.

3.3.6 Managing ResourceEntry Objects

Section 2.2.4, "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

```
//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

```
//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);
```

3.3.7 Managing Permission Sets

Section 2.3.4, "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.

3.3.8 Managing the Policy

Section 2.2.8, "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

```
// 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

```
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

- Obigation
- Attribute
- Function

For more information, see the Oracle Entitlements Server Java API Reference.

3.4 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 sames components can be managed at the `PolicyDomainEntry` scope if a `PolicyDomainEntry` has been created for further partitioning.

- Section 3.3.6, "Managing ResourceEntry Objects"
- Section 3.3.7, "Managing Permission Sets"
- Section 3.3.8, "Managing the Policy"

For information on using the `PolicyDomainEntry`, see Section 5.7, "Delegating with a Policy Domain."

4

Distributing Policies

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.

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.
- Perserving policy objects in a file-based persistent cache, local to the Policy Distribution Component, that provides independence from the policy store.

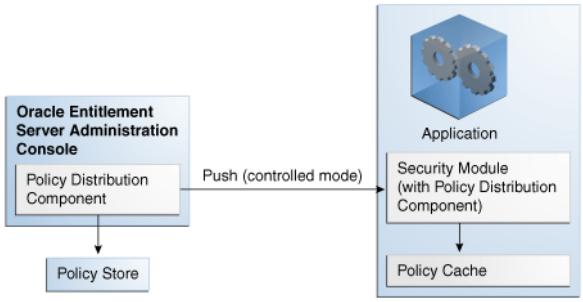
Both the central Oracle Entitlements Server Administration Console and the locally-installed (to the protected application) Security Module contain the Policy Distribution Component. This architecture allows two deployment scenarios: the first involves a centralized Policy Distribution Component that 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.

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.

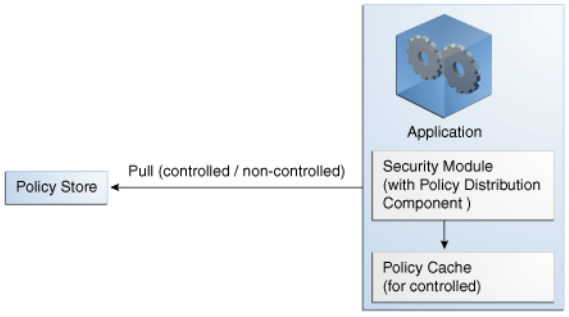
Figure 4–1 Using Oracle Entitlements Server Policy Distribution Component



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 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 the default distribution mode. It 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 can not 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.

With controlled 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. 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.

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.

With controlled distribution, if any policy distribution operation fails, the entire policy distribution fails. By default, controlled distribution is disabled.

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. Non-controlled distribution makes policy changes available as soon as they are saved to the policy store. Non-controlled distribution is initiated by the Security Module and may retrieve policies that are not yet complete. 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 (dependent on the mode of distribution) 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
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.

- Display Name - 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

```
//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

```
//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

```
//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

```
//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

```
//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

```
//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

```
//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 = pd.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.

> **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 progess) and includes a start time, an end time, the distribution initiater, and whether the distribution is successful or not for each PDP.

5 Delegating Policy Administration

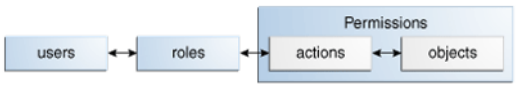
System administrative rights and policy management permissions can be delegated from one administrator to another by creating Administration Roles with restricted rights, or by granting an existing Administration Role to a user. Administration Roles consist of a subject (the person to whom the role is granted), the resources (the objects to which the role pertains) and actions (view, manage). This chapter documents information on how to delegate policy and system administrative tasks. It contains the following sections:

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, 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.

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 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.

> **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 5.4.1, "Creating An Administration Role."
2. Define the resource and appropriate actions as documented in Section 5.4.2, "Assigning Actions and Resources (Permissions) to an Administration Role."
3. Assign users (principals) as documented in Section 5.4.3, "Assigning Principals to an Administration Role."

Section 5.4.4, "Retrieving a Principal's Administration Resources" contains information on how to retrieve the administration roles that a principal has been assigned.

5.4.1 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.

```
//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.

5.4.2 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.

```
//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 5–1 Resource Name Options

Name	Description
ADMIN_POLICY	Allows management of Administration Role membership and permissions
ADMIN_ROLE	Allows management of Administration Roles
APPLICATION_ POLICY	Allows management of Application Policy objects
APPLICATION_ ROLE	Allows management of Application Roles
CONFIGURATION	Allows management of Security Modules
DISTRIBUTE_ APPLICATION_ POLICY	Allows administrator to initiate policy distribution
ENROLL	Allows administrator to enroll a Security Module instance
EXTENSION	Allows management of Functions and Attributes
PERMISSION_SET	Allows management of Permission Sets
POLICY	Allows management of Policies
RESOURCE_TYPE	Allows management of Resource Types
RESOURCE	Allows management of Resources
ROLE_CATEGORY	Allows management of Role Categories
SUB_POLICY_DOMAIN	Allows management of child Policy Domain objects

5.4.3 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.

```
//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.

5.4.4 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.)

5.5 Managing Administration Roles

Section 5.4, "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

```
//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

```
//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.

5.6 Using the Default Administration Roles

After installing Oracle Entitlements Server, 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.

5.7 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.

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.

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.

Caution: Deleting a `PolicyDomainEntry` object deletes all child objects created within it.

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."

- A `PermissionSetEntry` (one or more `ResourceActionsEntry` objects that associate a specific resource with the actions that can be performed on it). See Section 2.3.4, "Defining Permission Sets" for more information.
- A `PolicyEntry` (includes one `PolicyRuleEntry`, one `PermissionSetEntry`, one `PrincipalEntry` or `AppRoleEntry` and, optionally, one `ObligationEntry`). See Section 2.2.8, "Defining the Policy" for more information.

- An `AdminRoleEntry` (to define management of the domain). See Section 2.2.4, "Instantiating a Resource" for more information.

Note: The same target *Resource* can not be shared between Policy Domains.

6

Handling Authorization Calls and Decisions

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

6.1 Using the Authorization Request API

The Oracle Entitlements Server Authorization API are used by components for authorization checks during runtime. The following comprise the authorization API options in this release.

- **PEP API** - The AzAPI defines a set of interfaces that enable a Java module to supply and consume all the necessary information for submitting an eXtensible Access Control Markup Language (XACML) resquest and receiving a XACML response. The PEP API is a package built on top of the AzAPI package to simplify the creation of Policy Enforcement Points (PEP). See Section 6.2, "Using the PEP API" for more information.
- **`checkPermission()`** - This method uses Java `Permission` objects to grant access to protected resources. See Section 6.4, "Making checkPermission() Calls" for more information.

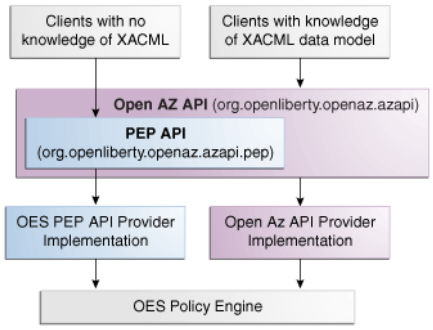
6.2 Using the PEP API

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 messages 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 resquest and receiving an authorization response.

Note: More information on the OpenAZ project can be found at `http://openliberty.org/wiki/index.php/OpenAz_Main_Page`.

The PEP API is a Java package built on top of the AzAPI. It contains utility classes for building a Policy Enforcement Point (PEP), and is designed to present a more simplified, scalable interface than the AzAPI, using native Java objects rather than XACML data objects. Figure 6–1 illustrates the relationship between the AzAPI, the PEP API and Oracle Entitlements Server.

Figure 6–1 Relationship Between Open AZ API and PEP API



Oracle Entitlements Server provides an implementation of the `org.openliberty.openaz.azapi.pep` package. The PEP API provider is packaged in `oracle.security.jps.openaz.pep`. For each PEP API request, the PEP API provider implementation is responsible for converting and mapping native Java objects to the underlying security platform. For each `PepRequest`, the PEP API provider invokes the Oracle Entitlements Server Authorization Engine and returns a `PepResponse` object. The provider also provides a default `DecisionHandler` implementation. The following sections have more information.

- Section 6.2.1, "Using the PEP API"
- Section 6.2.2, "Formatting PEP API Authorization Request Strings"
- Section 6.2.3, "Processing Query Requests"
- Section 6.2.4, "Getting Obligations"
- Section 6.2.5, "Configuring the PEP API"

6.2.1 Using the PEP API

This section contains sample code that illustrates ways in which the PEP API can be used. Example 6–1 shows how to authenticate the user with the login service and use the authentiated subject in a PEP API authorization request. This code is specific to a Java Standard Edition (JSE) container.

Example 6–1 Using Authenticated Subject in PEP API Request

```
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");

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 6–2 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 with 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 with 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 = " +
obligations.get(name).getStringValues());
        }
```

Note: It is recommended to call the `newPepRequest()` method with a Java Authentication and Authorization Service (JAAS) subject and not a string subject. A string subject will be converted to a JAAS subject.

6.2.2 Formatting PEP API Authorization Request Strings

Example 6–4 illustrates the `newQueryPepRequest()` method for creating an authorization request using subject and environment objects.

Example 6–4 newQueryPepRequest Method

```
public PepRequest newQueryPepRequest
    (object subjectObj,
    Object environmentObj
    String scope
    PepRequestQueryType queryType)
     throws PepException
```

This method contains a string to define the scope of the request. Within the scope string is defined a resource string. The following sections contain information on how to format these strings.

- Section 6.2.2.1, "Formatting the Scope String"
- Section 6.2.2.2, "Formatting the Resource String"

6.2.2.1 Formatting the 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:

```
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 6.2.2.2, "Formatting the Resource String."
- `actions` is optional. If present, it contains a comma separated list of requested actions.
- `searchscope` is optional 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 6.2.2.2, "Formatting the 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 6.2.2.2, "Formatting the 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";
```

6.2.2.2 Formatting the Resource String

The string should be in the format *appId* + / + *resourceType* + / + *resourceName*. The forward slash (*/*) is the delimiter. The `appId` and `resourceType` cannot be empty but the `resourceName` can be empty for a query request only.

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. The special characters `\` and `/` must be escaped as in the following examples:

- `myapp/computer\/laptop/mybox` signifies a resource in the application `myapp` with the resource type `computer/laptop` and the resource name `mybox`.
- `myapp/computer\\laptop/mybox` signifies a resource in the application `myapp` with the resource type `computer\laptop` and the resource name `mybox`.
- `myapp/computer\laptop/mybox` is invalid because the character after `\` is neither `/` nor `\`.

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

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

6.2.3 Processing Query Requests

The code in this section are examples of a query against a particular resource. Example 6–5 is a query request against a particular resource. Note that the search scope is defined as *immediate*.

Example 6–5 Requesting Authorization Against a Resource

```
...
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);
```

```
//this method is backed by AuthorizationService.queryActionsOnResource

PepResponse resp = req.decide();

//List of RuntimeAction objects
List actions = (List) resp.getAction();
RuntimeResource resource = (RuntimeResource) resp.getResource();
```

Example 6–6 is a query request against a particular resource and its children. Note that the search scope is defined as *children*.

Example 6–6 Requesting Authorization Against a Resource and Children

```
...
String scope = "resource=PepQueryTest/Hierarchical/\\/res1";

PepRequest req = PepRequestFactoryImpl.getPepRequestFactory
   (subject, env, scope, PepRequestQueryType.VERBOSE);

//this method is backed by AuthorizationService.queryActionsOnChildResource

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 6–7 is an example of code written for bulk authorization.

Example 6–7 Requesting Bulk Authorization

```
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();

//
// first request
//

        assertTrue(resp.next());

        assertTrue("resp.allowed() is expected to be true!! ", resp.allowed());
        assertSame(MY_ACTION, resp.getAction());
        assertSame(RESOURCE_STRING, resp.getResource());

//
// second request
//

        assertTrue(resp.next());

        assertFalse("resp.allowed() is expected to be false!! ", resp.allowed());
        assertSame(wrongAction, resp.getAction());
        assertSame(RESOURCE_STRING, resp.getResource());

//
// call next() again..
//
        assertFalse(resp.next());
    }
```

6.2.4 Getting 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.) Example 6–8 is an authorization request that also requests any Obligations.

Example 6–8 Getting Obligation with PEP API Authorization Request

```
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 + ", values = " +
obligations.get(name).getStringValues());         }
```

Example 6–9 is an example of an Obligation output. Again, there are no double quotes around the string value.

Example 6–9 Returning Obligations in a PEP API Response

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

6.2.5 Configuring the PEP API

To use the PEP API, the identity store, the policy store, the Policy Distribution Service, and the user assertion login module must be defined in `jps-config.xml`. Example 6–10 is not a complete file but copied below for demonstration purposes.

Example 6–10 Sample jps-config.xml File

```
...
<serviceInstance name="idstore.ldap" provider="idstore.ldap.provider">
 <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="pdp.service" provider="pdp.service.provider">
 <property name="sm_configuration_name" value="permissionSm"/>
 <property name="work_folder" value="/tmp"/>
 <property name="authorization_cache_enabled" value="true"/>
 <property name="role_cache_enabled" value="true"/>
 <property name="session_eviction_capacity" value="500"/>
 <property name="session_eviction_percentage" value="10"/>
 <property name="session_expiration_sec" value="60"/>
 <property name="oracle.security.jps.ldap.policystore.refresh.interval"
  value="30000"/>
 <property name="oracle.security.jps.policystore.refresh.purge.timeout"
  value="600000"/> <\!-\- 10 minutes -->
 <property name="loading_attribute_backward_compatible" value="false"/>
<\!-\- Properties for controlled mode PD -->
 <property name="oracle.security.jps.runtime.policy.distribution.mode"
  value="non-controlled"/>
 <property name="oracle.security.jps.runtime.configuration.id"
  value="${atzsrg.pdp.configuration_id}"/>
 <property name="oracle.security.jps.runtime.instance.name"
  value="${atzsrg.pdp.instance_name}"/>
 <property name="oracle.security.jps.runtime.flushDistribution"
  value="${atzsrg.pdp.always_flush}"/>
</serviceInstance>
<serviceInstance name="policystore.oid" provider="policy.oid">
 <property name="max.search.filter.length" value="4096"/>
 <property name="security.principal" value="cn=orcladmin"/>
 <property name="security.credential" value="welcome1"/>
 <property name="ldap.url" value="ldap://scl58116.us.oracle.com:3060"/>
 <property name="oracle.security.jps.ldap.root.name" value="cn=jpsTestNode"/>
 <property name="oracle.security.jps.farm.name" value="cn=duyang_wls.atzsrg"/>
 <property name="oracle.security.jps.policystore.resourcetypeenforcementmode"
  value="Lenient"/>
</serviceInstance>
<serviceInstance name="user.assertion.loginmodule" provider="jaas.login.provider">
 <description>User Assertion Login Module</description>
```

```
 <property name="loginModuleClassName"
  value="oracle.security.jps.internal.jaas.module.assertion.
  JpsUserAssertionLoginModule"/>
 <property name="jaas.login.controlFlag" value="REQUIRED"/>
</serviceInstance>
...
<jpsContexts default="default">
<jpsContext name="default">
<serviceInstanceRef ref="idstore.ldap"/>
<serviceInstanceRef ref="idstore.loginmodule"/>
<serviceInstanceRef ref="policystore.oid"/>
<serviceInstanceRef ref="pdp.service"/>
<serviceInstanceRef ref="audit"/>
<serviceInstanceRef ref="pip.service.ootb.ldap"/>
...
</jpsContext>
</jpsContexts>
```

6.3 Making XACML Calls

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 6–11 illustrates how to establish the section and send a XACML 2.0 authorization request.

Example 6–11 Sample Code to Establish Session For XACML Gateway

```
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.setRequestedCredentialType(OES_CREDENTIAL_TYPE);
        AuthenticationResultType result = atzSvc.establishSession(sess);
        identity = result.getIdentityAssertion();
      }
      catch (Exception e) {
        throw new XACMLException("Unable to authenticate user.", e);
      }
      if (identity == null) {
          throw new XACMLException("Null identity received.
             Unable to establish session for " + subject);
      }
      System.out.println("Authentication Succeeded, Identity: ");
      MessageElement ele = identity.get_any()[0];
      System.out.println(ele.getFirstChild());
  }
```

2. The PEP sends a XACML request containing the token to the PDP (Security Module). Example 6–12 is sample code that details how to create a XACML authorization request.

Example 6–12 Creating a XACML Request

```
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 6–13 is a sample XACML 2.0 authorization request. The `SSM-SOAPWS.wsdl` file provides the operation interface definitions.

Example 6–13 XACML 2.0 Authorization Request

```
<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"
     DataType="http://security.bea.com/ssmws/ssm-ws-1.0.wsdl#OESIdentityAssertion"
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"
      DataType="http://www.w3.org/2001/XMLSchema#string"
      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:Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
     DataType="http://www.w3.org/2001/XMLSchema#string"
      xsi:type="ns3:AttributeType">
  <ns3:AttributeValue
    xsi:type="ns3:AttributeValueType">borrow</ns3:AttributeValue>
  </ns3:Attribute>
  </ns3:Action>
```

```
  <ns4:Environment xsi:type="ns4:EnvironmentType"
     xmlns:ns4="urn:oasis:names:tc:xacml:2.0:context:schema:os"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ns4:Attribute AttributeId=
    "http://security.bea.com/ssmws/ssm-ws-1.0.wsdl#RegisteredAttribute"
     DataType="http://www.w3.org/2001/XMLSchema#string"
     xsi:type="ns4:AttributeType">
  <ns4:AttributeValue xsi:type="ns4:AttributeValueType">yes</ns4:AttributeValue>
    </ns4:Attribute>
    <ns4:Attribute AttributeId=
      "http://security.bea.com/ssmws/ssm-ws-1.0.wsdl
      #NumberOfBorrowedBooksAttribute"
       DataType="http://www.w3.org/2001/XMLSchema#string"
       xsi:type="ns4:AttributeType">
    <ns4:AttributeValue xsi:type="ns4:AttributeValueType">2</ns4:AttributeValue>
    </ns4:Attribute>
  </ns4:Environment>
</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 6–14 is a sample XACML 2.0 authorization response. The `SSM-SOAPWS.wsdl` file provides the operation interface definitions.

Example 6–14 XACML 2.0 Authorization Response

```
<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.wsdl#Roles" FulfillOn="Permit">
        <ns1:AttributeAssignment
          DataType="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="http://security.bea.com/ssmws/ssm-ws-1.0.wsdl#role">
          AuthenticatedUser</ns1:AttributeAssignment>
        </ns1:Obligation>
        <ns1:Obligation
          ObligationId="http://security.bea.com/ssmws/ssm-ws-1.0.wsdl#
          ResponseAttributes" FulfillOn="Permit">
        <ns1:AttributeAssignment
          DataType="http://www.w3.org/2001/XMLSchema#dateTime"
         AttributeId="http://security.bea.com/ssmws/ssm-ws-1.0.wsdl#decisionTime">
          2010-11-02T12:50:43.685Z</ns1:AttributeAssignment>
        </ns1:Obligation>
      </ns1:Obligations>
    </Result>
  </Response>
```

6.4 Making checkPermission() Calls

A Java `Permission` object represents access to a resource. A `Permission` object is constructed and assigned (access granted) based on the configured policy in effect. Java `checkPermission()` authorization is based on these permissions.

Oracle Entitlements Server supports the use of the `checkPermission()` method in the following standard classes:

- `java.lang.SecurityManager`
- `java.security.AccessController`

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

`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. The following sample illustrates how you might use `checkPermission()`.

```
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;
}
```

7 Extending Functionality

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"

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 `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 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.

7.1.2 Creating Custom Attribute Retrievers

As described in 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 `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 `com.bea.security.providers.authorization.asi.AttributeRetrieverV2` interface.

   See 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 retriever.

   Make sure the configuration specifies the fully-qualified location of the custom attribute retriever. See the Security Module configuration appendix in the *Oracle Fusion Middleware Administrator'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 7–1 Methods in AttributeRetrieverV2 Interface

Method	Description
`getAttributeValue()`	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: ■ Name defines the name of the attribute being retrieved. ■ 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. ■ Subject defines the user associated with the request. ■ Roles defines any role membership of the subject, or null if this is a role mapping call. ■ Resource defines the protected resource associated with the request. ■ contextHandler is the context associated with the request; this may be null if non-existent.
`getHandled AttributeNames()`	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. If the method returns a null or empty value, this attribute retriever object will be called when any dynamic attribute has to be resolved.

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

```
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

```
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.

7.2 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.

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. The following procedure details the steps to take when implementing a custom function in your policy.

1. Write the custom code for the function.
2. Compile the Java code.
3. Add the complied class to the class path of the Security Module.

Example 7–3 illustrates how you might create a custom function.

Example 7–3 Sample Code for a Custom Function

```
package com.bea.security.examples;

import java.util.Map;
import java.util.Properties;

import javax.security.auth.Subject;

import weblogic.security.service.ContextHandler;
import weblogic.security.spi.Resource;

import com.bea.security.providers.authorization.asi.ARME.evaluator.RequestHandle;
import com.wles.util.AttributeElement;

public class MyEvaluationFunction {

    /**named evaluation function. Additional authorization request data
     is made available to allow for more complex attribute evaluation.
     This method will be registered to ARME, and be invoked while the policy
     contains a custom evaluation function with name "string_longer_then".
     @param requestHandle the 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 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-existant
     @return <code>true</code> or <code>false</code> as the result of the function
     @throws MissingAttributeException for can not get required attribute value.
     */
    public boolean string_longer_then(RequestHandle requestHandle,
            Object[] args,
            Subject subject,
            Map roles,
            Resource resource,
            ContextHandler contextHandler) {
        // Check if we got a correct number of the input paramters
        if(args.length < 2 || args.length > 3 ||
                args[0] == null || args[1] == null) {
            // Incorrect number of arguments.
            // Such policy is invalid and can not be evaluated
            throw new RuntimeException
               ("Incorrect number of arguments in a function");
        }

        // Arguments for an evalaution function are attribute names.
        // Unfortunately, if a string literal or a numeric value is used
        // 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.

        // Get the integer value of the first argument.
        // This example does not do any type error checking
        // - but your code should.
```

```
        int intCompLength = 0;
        try {
            AttributeElement strLength =
requestHandle.getAttribute((String)args[0], true);
            if(strLength != null) {
                if(strLength.isList()) {
                    // string_longer_then: first argument not a single value
                    return false;
                }
                intCompLength =
Integer.parseInt((String)strLength.getValueAs(String.class));
            } else {
                // numerical constant will be passed in as is.
                try {
                    intCompLength = Integer.parseInt((String)args[0]);
                } catch(NumberFormatException ne) {
                    //value format is error, and no attribute in requestHandle.
                    throw new RuntimeException("miss attribute: " + args[0]);
                }
            }
        } catch(Exception e) {
            //caught exception while get attribute
            throw new RuntimeException(
                    "failed while get attribute: " + (String)args[0] + ".
Exception: " + e.getMessage());
        }

        // Get the string value
        String input = null;
        try {
            AttributeElement strContent =
requestHandle.getAttribute((String)args[1], true);
            if(strContent != null) {
                if(strContent.isList()) {
                    // string_longer_then: second argument is not a single value
                    return false;
                }
                input = (String)strContent.getValueAs(String.class);
            } else {
                input = (String)args[1];
            }
        } catch(Exception e) {
            //caught exception while get attribute
            throw new RuntimeException(
                    "failed while get attribute: " + (String)args[0] + ".
Exception: " + e.getMessage());
        }

        // return false, if the conditin is not satisfied
        if(input.length() <= intCompLength) {
            return false;
        }

        // Condition was satisfied.  Create and attach the return attribute

        /* The method appendReturnData(String name, Object data) on RequestHandle
object
           does a copy of data.
           Return value will only be appended if the rule
           that called this function actually fired.
```

```
            (other elements in the condition may prevent that)
        */
        requestHandle.appendReturnData("cropped_string", input);
        return true;
    }

}
```

Example 7–4 illustrates how you might create a custom function that returns a DataType argument.

Example 7–4 Sample Code for a Custom Function Returning DataType Argument

```
import oracle.security.jps.service.policystore.info.DataType;
import oracle.security.jps.service.policystore.info.OpssString;

 /**
     Another example of evaluation function.
     Additional authorization request data
     is made available to allow for more complex attribute evaluation.
     This method will be registered to ARME, and be invoked while the policy
     contains a custom evaluation function with name "string_longer_then".
     @param requestHandle the 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 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-existant
     @return <code>DataType</code> as the result of the function where
      DataType is any of the out-of-the-box supported data typs such as
      OpssDate/ OpssTime/ OpssString etc.
     @throws MissingAttributeException for can not get required attribute value.
     */
    public DataType string_to_upper_case(RequestHandle requestHandle,
            Object[] args,
            Subject subject,
            Map roles,
            Resource resource,
            ContextHandler contextHandler)
    throws MissingAttributeException {
        // Check if we got a correct number of the input paramters
        if((args.length != 1) || (args[0] == null)) {
            // Incorrect number of arguments.
            // Such policy is invalid and can not be evaluated
            throw new RuntimeException
               ("Incorrect number of arguments in a function");
        }

        // Arguments for an evalaution function are attribute names.
        // Unfortunately, if a string literal or a numeric value is used
        // 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.
```

```
        // Get the integer value of the first argument.
        // This example does not do any type error checking
        // - but your code should.
        int intCompLength = 0;
        try {
            AttributeElement str = requestHandle.getAttribute((String)args[0],
true);
            String input = null;
            if(str != null) {
                if(str.isList()) {
                    // string_to_upper_case: first argument not a single value
                    return false;
                }
                input = (String)str.getValueAs(String.class);
            } else {
                // string constant will be passed in as is.
                input = (String)args[0];
            }
        } catch(Exception e) {
            //caught exception while get attribute
            throw new RuntimeException(
                    "failed while get attribute: " + (String)args[0] + ".
Exception: " + e.getMessage());
        }


        String output = input.toUpperCase();

        return new OpssString(output);
    }
```

For more information, see Section 2.3.3.2, "Creating Custom Function Definitions" and Section 2.3.5, "Defining a Condition."

8

Using the JSP Tags

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, "Defining the Functional Tags"
- Section 8.2, "Defining the Assistant Tags"

> **Note:** The tag library can only be run on WebLogic Server.

8.1 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

8.1.1 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.

> **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 8–1 documents the `isAccessAllowed` tag definition.

Table 8–1 isAcessAllowed Tag Definition

Name	Details
resource	**Description:** The resource used when calling `isAccessAllowed`. **Mandatory** **Return Type:** not applicable
resourceType	**Description:** The type of resource used when calling `isAccessAllowed`. If not set, the global resource type set by `setSecurityContext` will be used. **Optional** **Return Type:** not applicable
action	**Description:** The action used when calling `isAccessAllowed`. The default action is view. **Optional** **Return Type:** not applicable
resultVar	**Description:** The name of the scripting variable used to tell if access is allowed. **Optional** **Return Type:** boolean
resultVarScope	**Description:** The scope of the `resultVar` (page, request, session, or application). The default scope is page. **Optional** **Return Type:** not applicable
obligationVar	**Description:** The name of the variable used for returning obligations from the `isAccessAllowed` call. **Optional** **Return Type:** A map of obligations; the key is the obligation name and the value is a map of attributes with attribute names and values.
obligationVarScope	**Description:** The scope of the variable containing obligations from `isAccessAllowed` (page, request, session, or application). The default scope is page. **Optional** **Return Type:** not applicable

Example 8–1 illustrates how `isAccessAllowed` may be used.

Example 8–1 isAccessAllowed Tag Example

```
<%-- Set global attributes --%>
  <oes:setSecurityContext appId="TagLibraryApp" resourceType="image"
     resourcePrefix="images/">
    <oes:attribute name="test_attr" value="good_job"/>
  </oes:setSecurityContext>

  <%!
      String resourceStr="private.jpg";
      String actionStr="read";
      String returnVar = "isAllowed";
  %>

<%-- Test for isAccessAllowed tag --%>
  <oes:isAccessAllowed  resource="<%=resourceStr %>" action="<%=actionStr %>"
  resultVar="<%=returnVar %>" obligationVar="obligations">
```

```
    <oes:attribute name="test_attr_local" value="hard_work" />
    <oes:then>
         You have the permission to <%=actionStr %> the image <%=resourceStr %>.
         <br/>
         <img src="images/private.jpg"  width="250" height="150" /> <br/>
          The obligations are: <br/>
            <c:forEach items="${obligations}" var="entry">
            <c:out value="${entry.key}" />  =  <c:out
               value="${entry.value}" /> <br/>
            </c:forEach>
    </oes:then>
    <oes:else>
         You have not the permission to <%=actionStr %> the image
          <%=resourceStr %>. <br/>
    </oes:else>
  </oes:isAccessAllowed>

<%-- another way to use tag isAccessAllowed --%>
  <oes:isAccessAllowed  resource="<%=resourceStr %>" action="<%=actionStr %>"
  resultVar="<%=returnVar %>" obligationVar="obligations">
    <oes:attribute name="test_attr_local" value="hard_work" />
  </oes:isAccessAllowed>
  <c:choose>
  <c:when test="${isAllowed}">You have the permission to <%=actionStr %>
      the image <%=resourceStr %>. <br/>
      <img src="images/private.jpg"  width="250" height="150" />
      The obligations are: <br/>
       <c:forEach items="${obligations}" var="entry">
       <c:out value="${entry.key}" />  = 
       <c:out value="${entry.value}" /> <br/>
  </c:forEach>
  </c:when>
 <c:otherwise>
      You have not the permission to <%=actionStr %> the image
         <%=resourceStr %>. <br/>
 </c:otherwise>
 </c:choose>
```

8.1.2 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.

Table 8–2 isAccessNotAllowed Tag Definition

Name	Details
resource	**Description:** The resource used when calling `isAccessAllowed`. **Mandatory** **Return Type:** not applicable

Table 8–2 (Cont.) isAccessNotAllowed Tag Definition

Name	Details
resourceType	**Description:** The type of resource used when calling `isAccessAllowed`. If not set, the global resource type set by `setSecurityContext` will be used. **Optional** **Return Type:** not applicable
action	**Description:** The action used when calling `isAccessAllowed`. The default action is view. **Optional** **Return Type:** not applicable
resultVar	**Description:** The name of the scripting variable used to tell if access is allowed. **Optional** **Return Type:** boolean
resultVarScope	**Description:** The scope of the `resultVar` (page, request, session, or application). The default scope is page. **Optional** **Return Type:** not applicable
obligationVar	**Description:** The name of the variable used for returning obligations from the `isAccessAllowed` call. **Optional** **Return Type:** A map of obligations. The key is the obligation name and the value is a map of attributes with attribute names and values.
obligationVarScope	**Description:** The scope of the variable containing obligations from `isAccessAllowed` (page, request, session, or application). The default scope is page. **Optional** **Return Type:** not applicable

Example 8–2 illustrates how `isAccessNotAllowed` may be used.

Example 8–2 isAccessNotAllowed Tag Example

```
<%-- Test for isAccessNotAllowed tag --%>
     <oes:isAccessNotAllowed  resource="<%=resourceStr %>"
      action="<%=actionStr %>" resultVar="isNotAllowed"
      obligationVar="obligations_not">
     <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}" />  = 
       <c:out value="${entry.value}" /> <br/>
       </c:forEach>
      </oes:then>
      <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}" />  = 
       <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="${isNotAllowed}">You have not the permission to
      <%=actionStr %> the image <%=resourceStr %>. <br/>
   </c:when>
   <c:otherwise>
    You have the permission to <%=actionStr %> the image <%=resourceStr %>. <br/>
    <img src="images/private.jpg"  width="250" height="150" />
    The obligations are: <br/>
   <c:forEach items="${obligations}" var="entry">
   <c:out value="${entry.key}" />  = 
   <c:out value="${entry.value}" /> <br/>
   </c:forEach>
   </c:otherwise>
   </c:choose>
```

8.1.3 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 8–3 getUserRoles Tag Definition

Name	Details
resource	**Description:** The resource used when calling `getUserRoles`. **Mandatory** **Return Type:** not applicable
resourceType	**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
action	**Description:** The action used when calling `getUserRoles`. The default action is view. **Optional** **Return Type:** not applicable
resultVar	**Description:** The name of the variable to set that contains the list of user's roles. **Mandatory** **Return Type:** A list of strings of role names.

Table 8–3 (Cont.) getUserRoles Tag Definition

Name	Details
resultVarScope	**Description:** The scope of the `resultVar` (page, request, session, or application). The default scope is page. **Optional** **Return Type:** not applicable

Example 8–3 illustrates how `getUserRoles` may be used.

Example 8–3 getUserRoles Tag Example

```
<%-- 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>
```

8.1.4 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.

> **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.

Table 8–4 isUserInRole Tag Definition

Name	Details
role	**Description:** The name of the role to check against the user. **Mandatory** **Return Type:** not applicable
resource	**Description:** The name of the resource against which to check the user's roles. **Mandatory** **Return Type:** not applicable
resourceType	**Description:** The type of resource against which to check the user's roles. If it is not set, the global resource type set by `setSecurityContext` will be used. **Optional** **Return Type:** not applicable

Table 8–4 (Cont.) isUserInRole Tag Definition

Name	Details
action	**Description:** The resource's action against which the user's role will be checked. The default value will be view. **Optional** **Return Type:** not applicable
resultVar	**Description:** A variable used to hold the result from `isUserInRole` for later use. **Optional** **Return Type:** boolean
resultVarScope	**Description:** The scope of the `resultVar` (page, request, session, or application). The default scope is page. **Optional** **Return Type:** not applicable

Example 8–4 illustrates how `isUserInRole` may be used.

Example 8–4 isUserInRole Tag Example

```
<%-- 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 can not see the content of
          protected/rolepolicy.jsp
       </c:otherwise>
       </c:choose
```

8.2 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

8.2.1 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). 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 8.2.2, "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 8–5 setSecurityContext Tag Definition

Name	Details
appId	**Description:** 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. **Mandatory** **Return Type:** not applicable
resourceType	**Description:** The global resource type which can be used by all other authorization tags. **Optional** **Return Type:** not applicable
resourcePrefix	**Description:** 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`. **Optional** **Return Type:** not applicable

Example 8–5 illustrates how `setSecurityContext` may be used.

Example 8–5 setSecurityContext Tag Example

```
<%-- Set global attributes --%>
    <oes:setSecurityContext appId="TagLibraryApp" resourceType="image"
      resourcePrefix="images/">
      <oes:attribute name="test_attr" value="good_job"/>
    </oes:setSecurityContext>
```

8.2.2 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 8–6 attribute Tag Definition

Name	Details
name	**Description:** The name of the attribute to set in the application context. **Mandatory** **Return Type:** not applicable
value	**Description:** The value of the attribute to set in the application context. **Mandatory** **Return Type:** not applicable

Example 8–6 illustrates how `attribute` may be used.

Example 8–6 attribute Tag Example

```
<oes:attribute name="myroleattr" value="its_my_role"/>
```

8.2.3 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 defintion.

9

Enhancing the Development Environment

Oracle Entitlements Server provides logging to enhance the developer's environment.

9.1 Logging

Oracle Entitlements Server uses the standard Java package `java.util.logging` for logging. The name of the logging setup file is `logging.properties`. It is the standard configuration file for the Java Development Kit (JDK) and it is located (by default) in `$JAVA_HOME/jre/lib/`.

> **Note:** Configure the location of `logging.properties` by running the following on the command line with the actual path based on your install.
>
> `-Djava.util.logging.config.file=`*`/path/filename`*

Java logging defines log levels to control output ranging from FINEST (the lowest priority with the least amount of detail) to SEVERE (the highest priority intended for fatal program errors and the like). Enabling logging at a given level also enables logging at all higher levels. Table 9–1 contains the specific logger properties that can be set to FINE (details for debugging and diagnosing problems) to provide information for purposes of troubleshooting server issues.

Table 9–1 Logging Server Issues

To Troubleshoot...	Set These Properties To FINE
Policy management issues	▪ `oracle.jps.policymgmt`
Basic authorization issues	▪ `oracle.jps.authorization`
Policy distribution issues	▪ `oracle.oes.pd` ▪ `oracle.jps.common`
WebLogic Server, RMI or Web Services Security Module issues	▪ `oracle.oes.sm` ▪ `oracle.jps.common`

After modifying `logging.properties`, ensure the `java.util.logging.config.file` system property is set by running the following command:

`-Djava.util.logging.config.file=/<`*`directory`*`>/logging.properties`

Example 9–1 is the default Oracle Entitlements Server `logging.properties` file.

Example 9–1 Default logging.properties Configuration File

```
############################################################
#  Default Logging Configuration File
# You can use a different file by specifying a filename
# with the java.util.logging.config.file system property.
# For example java -Djava.util.logging.config.file=myfile
############################################################
#  Global properties
############################################################
# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.
handlers= java.util.logging.ConsoleHandler, java.util.logging.FileHandler
# To also add the FileHandler, use the following line instead.
#handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler
# Default global logging level.
# This specifies which kinds of events are logged across
# all loggers. For any given facility this global level
# can be overriden by a facility specific level
# Note that the ConsoleHandler also has a separate level
# setting to limit messages printed to the console.
.level= WARNING
############################################################
# Handler specific properties.
# Describes specific configuration info for Handlers.
############################################################
# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = /logs/java%u.log
java.util.logging.FileHandler.limit = 50000000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
############################################################
# Facility specific properties.
############################################################
# Provides extra control for each logger.
############################################################
# For example, set the com.xyz.foo logger to only log SEVERE
# messages:
oracle.jps.authorization = FINE
oracle.oes.sm=FINE
oracle.oes.common=FINE
oracle.oes.tool=FINE
oracle.oes.pd=FINE
oracle.jps.policymgmt=FINE
oracle.jps.authorization=FINE
oracle.jps.common=FINE
```

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V