Oracle® Fusion Middleware

Developing Security Providers for Oracle WebLogic Server

15c (15.1.1.0.0)
G32024-01
October 2025

ORACLE

Oracle Fusion Middleware Developing Security Providers for Oracle WebLogic Server, 15c (15.1.1.0.0)

G32024-01

Primary Author: Oracle Corporation

Contents

3 Authentication Providers

4 Identity Assertion Providers

9 Auditing Providers

10 Credential Mapping Providers

11 Auditing Events From Custom Security Providers

12 Servlet Authentication Filters

13 Versionable Application Providers

14 CertPath Providers

A MBean Definition File (MDF) Element Syntax

B Generate an MBean Type Using the WebLogic MBeanMaker

Preface

This document provides security vendors and application developers with the information needed to develop new security providers for use with WebLogic Server.

Audience

This document is written for independent software vendors (ISVs) who want to write their own security providers for use with WebLogic Server. It is assumed that most ISVs reading this documentation are sophisticated application developers who have a solid understanding of security concepts, and that no basic security concepts require explanation. It is also assumed that security vendors and application developers are familiar with WebLogic Server and with Java (including Java Management eXtensions (JMX)).

Documentation Accessibility

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

Access to Oracle Support

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

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documentation

The Oracle corporate Web site provides all documentation for WebLogic Server. Other WebLogic Server documents that may be of interest to security vendors and application developers working with security providers are:

- *Understanding Security for Oracle WebLogic Server*
- *Administering Security for Oracle WebLogic Server*
- *Developing Applications with the WebLogic Security Service*

- *Securing Resources Using Roles and Policies for Oracle WebLogic Server*
- *Securing a Production Environment for Oracle WebLogic Server*
- Java API Reference for Oracle WebLogic Server

New and Changed WebLogic Server Features

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic 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

Introduction to Developing Security Providers for WebLogic Server

This chapter prepares you to learn more about developing security providers.
This chapter includes the following sections:

Prerequisites for This Guide

Prior to reading this guide, you should review the following sections in *Understanding Security for Oracle WebLogic Server*:

- Security Providers
- WebLogic Security Framework

Additionally, WebLogic Server security includes many unique terms and concepts that you need to understand. These terms and concepts—which you will encounter throughout the WebLogic Server security documentation—are defined in Security Fundamentals in *Understanding Security for Oracle WebLogic Server*.

Overview of the Development Process

This section is a high-level overview of the process for developing new security providers, so you know what to expect. Details for each step are discussed later in this guide.

The main steps for developing a custom security provider are:

- Designing the Custom Security Provider
- Creating Runtime Classes for the Custom Security Provider by Implementing SSPIs
- Generating an MBean Type to Configure and Manage the Custom Security Provider
- Configuring the Custom Security Provider
- Providing Management Mechanisms for Security Policies, Security Roles, and Credential Maps

Designing the Custom Security Provider

The design process includes the following steps:

1. Review the descriptions of the WebLogic security providers to determine whether you need to create a custom security provider.

   Descriptions of the WebLogic security providers are available under WebLogic Security Providers in *Understanding Security for Oracle WebLogic Server* and in later sections of this guide under the Do You Need to Create a Custom <*Provider_Type*> Provider? headings.

2. Determine which type of custom security provider you want to create.

   The type may be authentication, identity assertion, principal validation, authorization, adjudication, role mapping, auditing, credential mapping, versionable application, or CertPath, as described in Types of Security Providers in *Understanding Security for Oracle WebLogic Server*. Your custom security provider can augment or replace the WebLogic security providers that are already supplied with WebLogic Server.

3. Identify which security service provider interfaces (SSPIs) you must implement to create the runtime classes for your custom security provider, based on the type of security provider you want to create.

   The SSPIs for the different security provider types are described in Security Services Provider Interfaces (SSPIs) and summarized in SSPI Quick Reference.

4. Decide whether you will implement the SSPIs in one or two runtime classes.

   These options are discussed in Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes .

5. Identify which required SSPI MBeans you must extend to generate an MBean type through which your custom security provider can be managed. If you want to provide additional management functionality for your custom security provider (such as handling of users, groups, security roles, and security policies), you also need to identify which optional SSPI MBeans to implement.

   The SSPI MBeans are described in Security Service Provider Interface (SSPI) MBeans and summarized in SSPI MBean Quick Reference.

6. Determine how you will initialize the database that your custom security provider requires. You can have your custom security provider create a simple database, or configure your custom security provider to use an existing, fully-populated database.

   These two database initialization options are explained in Initialization of the Security Provider Database.

7. Identify any database seeding that your custom security provider will need to do as part of its interaction with security policies on WebLogic resources. This seeding may involve creating default groups, security roles, or security policies.

   See Security Providers and WebLogic Resources.

Creating Runtime Classes for the Custom Security Provider by Implementing SSPIs

In one or two runtime classes, implement the SSPIs you have identified by providing implementations for each of their methods. The methods should contain the specific algorithms for the security services offered by the custom security provider. The content of these methods describe how the service should behave.

Procedures for this task are dependent on the type of security provider you want to create, and are provided under the Create Runtime Classes Using the Appropriate SSPIs heading in the sections that discuss each security provider in detail.

Generating an MBean Type to Configure and Manage the Custom Security Provider

Generating an MBean type includes the following steps:

1. Create an MBean Definition File (MDF) for the custom security provider that extends the required SSPI MBean, implements any optional SSPI MBeans, and adds any custom attributes and operations that will be required to configure and manage the custom security provider.

   Information about MDFs is available in Understand the Basic Elements of an MBean Definition File (MDF), and procedures for this task are provided under the Create an MBean Definition File (MDF) heading in the sections that discuss each security provider in detail.

2. Run the MDF through the WebLogic MBeanMaker to generate intermediate files (including the MBean interface, MBean implementation, and MBean information files) for the custom security provider's MBean type.

   Information about the WebLogic MBeanMaker and how it uses the MDF to generate Java files is provided in Understand What the WebLogic MBeanMaker Provides , and procedures for this task are provided under the Use the WebLogic MBeanMaker to Generate the MBean Type heading in the sections that discuss each security provider in detail.

3. Edit the MBean implementation file to supply content for any methods inherited from implementing optional SSPI MBeans, as well as content for the method stubs generated as a result of custom attributes and operations added to the MDF.

4. Run the modified intermediate files (for the MBean type) and the runtime classes for your custom security provider through the WebLogic MBeanMaker to generate a JAR file, called an MBean JAR File (MJF).

   Procedures for this task are provided under the Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF) heading in the sections that discuss each security provider in detail.

5. Install the MBean JAR File (MJF) into the WebLogic Server environment.

   Procedures for this task are provided under the Install the MBean Type into the WebLogic Server Environment heading in the sections that discuss each security provider in detail.

Configuring the Custom Security Provider

> **Note**
>
> The configuration process can be completed by the same person who developed the custom security provider, or by a designated administrator.

The configuration process consists of using the WebLogic Remote Console to supply the custom security provider with configuration information. If you generated an MBean type for managing the custom security provider, configuring the custom security provider in the WebLogic Remote Console also means that you are creating a specific instance of the MBean type.

See *Administering Security for Oracle WebLogic Server*.

Providing Management Mechanisms for Security Policies, Security Roles, and Credential Maps

Certain types of security providers need to provide administrators with a way to manage the security data associated with them. For example, an authorization provider needs to supply administrators with a way to manage security policies. Similarly, a role mapping provider needs to supply administrators with a way to manage security roles, and a credential mapping provider needs to supply administrators with a way to manage credential maps.

For the WebLogic Authorization, Role Mapping, and Credential Mapping providers, there are already management mechanisms available for administrators in the WebLogic Remote Console. However, do you not inherit these mechanisms when you develop a custom version of one of these security providers; you need to provide your own mechanisms to manage security policies, security roles, and credential maps. These mechanisms must read and write the appropriate security data to and from the custom security provider's database, but may or may not be integrated with the WebLogic Remote Console.

For more information, refer to one of the following sections:

- Provide a Mechanism for Security Policy Management (for custom authorization providers)
- Provide a Mechanism for Security Role Management (for custom role mapping providers)
- Provide a Mechanism for Credential Map Management (for custom credential mapping providers)

2

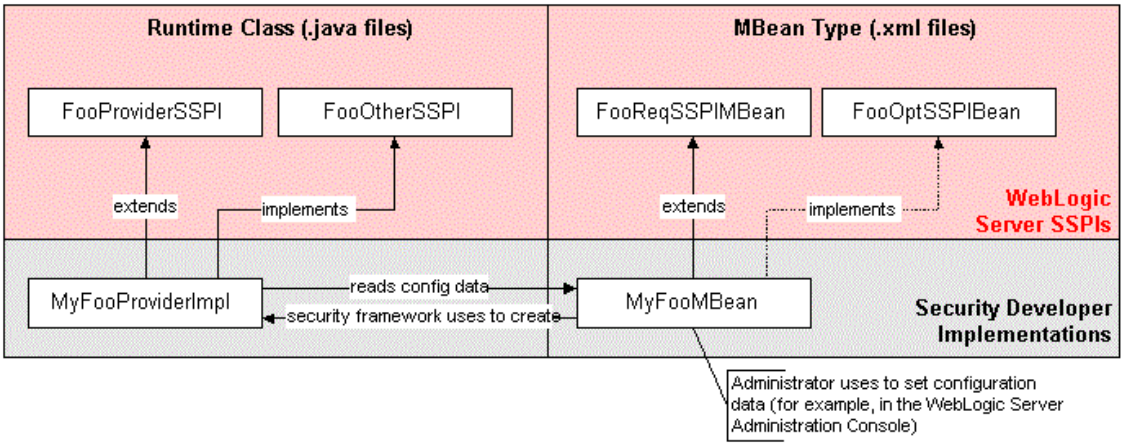
Design Considerations

This chapter describes security provider concepts and functionality in more detail to help you get started. Careful planning of development activities can greatly reduce the time and effort you spend developing custom security providers.
This chapter includes the following sections:

General Architecture of a Security Provider

Although there are different types of security providers you can create (see Types of Security Providers in *Understanding Security for Oracle WebLogic Server*), all security providers follow the same general architecture. Figure 2-1 illustrates the general architecture of a security provider, and an explanation follows.

Figure 2-1 Security Provider Architecture



> **Note**
>
> The SSPIs and the runtime classes (that is, implementations) you will create using the SSPIs are shown on the left side of Figure 2-1 and are .java files.
>
> Like the other files on the right side of Figure 2-1, MyFooMBean begins as a .xml file, in which you will extend (and optionally implement) SSPI MBeans. When this MBean Definition File (MDF) is run through the WebLogic MBeanMaker utility, the utility generates the .java files for the MBean type, as described in Generating an MBean Type to Configure and Manage the Custom Security Provider .

Figure 2-1 shows the relationship between a single runtime class (`MyFooProviderImpl`) and an MBean type (`MyFooMBean`) you create when developing a custom security provider. The process begins when a WebLogic Server instance starts, and the WebLogic Security Framework:

1. Locates the MBean type associated with the security provider in the security realm.
2. Obtains the name of the security provider's runtime class (the one that implements the Provider SSPI, if there are two runtime classes) from the MBean type.
3. Passes in the appropriate MBean instance, which the security provider uses to initialize (read configuration data).

Therefore, both the runtime class (or classes) *and* the MBean type form what is called the security provider.

Security Services Provider Interfaces (SSPIs)

As described in Overview of the Development Process, you develop a custom security provider by first implementing a number of security services provider interfaces (SSPIs) to create runtime classes. This section helps you:

- Understand Two Important Restrictions
- Understand the Purpose of the Provider SSPIs
- Understand the Purpose of the Bulk Access Providers
- Determine Which Provider Interface You Will Implement
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

Additionally, this section provides SSPI Quick Reference that indicates which SSPIs can be implemented for each type of security provider.

Understand Two Important Restrictions

Security providers must adhere to the following restrictions:

- A custom security provider's runtime class implementation must not contain any code that requires a security check to be performed by the WebLogic Security Framework. Doing so causes infinite recursion, because the security providers are the components of the WebLogic Security Framework that actually perform the security checks and grant access to WebLogic resources.

- No local (where local refers to the same server, cluster, or domain) Java Platform, Enterprise Edition (Java EE) services are available for use within a security provider's implementation. Any attempt to use them is unsupported. For example, this prohibits calling an EJB in the current domain from your security provider.

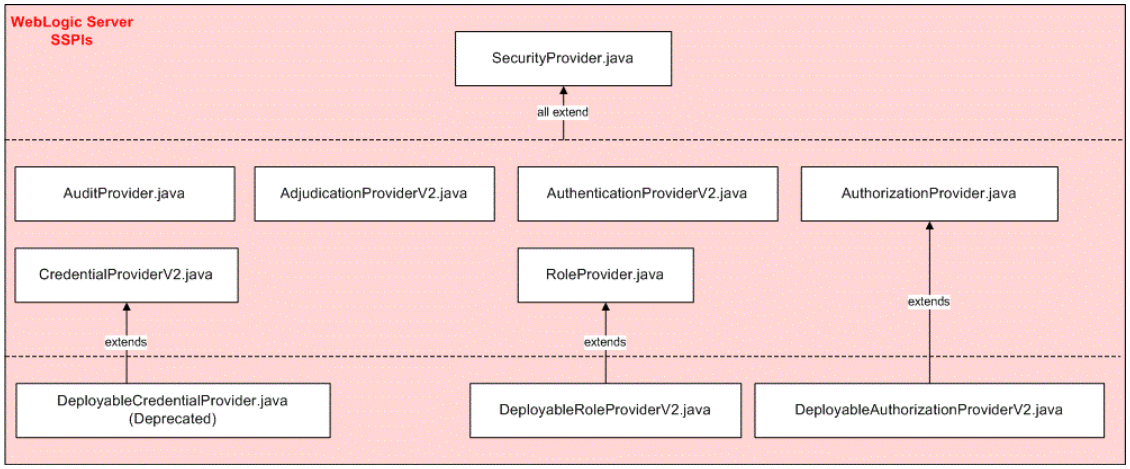
  Java EE services in other domains are accessible and can be used within a security provider.

> **Note**
>
> When writing your custom security provider, use only public WebLogic Server classes as documented in *Java API Reference for Oracle WebLogic Server*. Do not use or extend internal WebLogic Server or WebLogic Security provider classes. Doing so can cause unexpected results.

Understand the Purpose of the Provider SSPIs

Each SSPI that ends in the suffix Provider (for example, `CredentialProvider`) exposes the services of a security provider to the WebLogic Security Framework. This allows the security provider to be manipulated (initialized, started, stopped, and so on).

Figure 2-2 "Provider" SSPIs



As shown in Figure 2-2, the SSPIs exposing security services to the WebLogic Security Framework are provided by WebLogic Server, and all extend the `SecurityProvider` interface, which includes the following methods:

- initialize

  ```
  public void initialize(ProviderMBean providerMBean, SecurityServices
  securityServices)
  ```

  The `initialize` method takes as an argument a `ProviderMBean`, which can be narrowed to the security provider's associated MBean instance. The MBean instance is created from the MBean type you generate, and contains configuration data that allows the custom security provider to be managed in the WebLogic Server environment. If this configuration data is available, the `initialize` method should be used to extract it.

The `securityServices` argument is an object from which the custom security provider can obtain and use the Auditor Service. See Auditing Providers and Auditing Events From Custom Security Providers.

- getDescription

  ```
  public String getDescription()
  ```

  This method returns a brief textual description of the custom security provider.

- shutdown

  ```
  public void shutdown()
  ```

  This method shuts down the custom security provider.

Because they extend `SecurityProvider`, a runtime class that implements any SSPI ending in `Provider` must provide implementations for these inherited methods.

Understand the Purpose of the Bulk Access Providers

This release of WebLogic Server includes bulk access versions of the following authorization, adjudication, and role mapping provider SSPI interfaces:

- `BulkAuthorizationProvider`
- `BulkAccessDecision`
- `BulkAdjudicationProvider`
- `BulkAdjudicator`
- `BulkRoleProvider`
- `BulkRoleMapper`

The bulk access SSPI interfaces allow authorization, adjudication, and role mapping providers to receive multiple decision requests in one call rather than through multiple calls, typically in a `'for'` loop. The intent of the bulk SSPI variants is to allow provider implementations to take advantage of internal performance optimizations, such as detecting that many of the passed-in `Resource` objects are protected by the same policy and will generate the same decision result.

See Bulk Authorization Providers, Bulk Adjudication Providers, and Bulk Role Mapping Providers for additional information.

Determine Which Provider Interface You Will Implement

Implementations of SSPIs that begin with the prefix `Deployable` and end with the suffix `Provider` (for example, `DeployableRoleProviderV2`) expose the services of a custom security provider into the WebLogic Security Framework as explained in Understand the Purpose of the Provider SSPIs. However, implementations of these SSPIs also perform additional tasks. These SSPIs also provide support for security in deployment descriptors, including the servlet deployment descriptors (`web.xml`, `weblogic.xml`), the EJB deployment descriptors (`ejb-jar.xml`, `weblogic-ejb.jar.xml`) and the EAR deployment descriptors (`application.xml`, `weblogic-application.xml`).

Authorization providers, role mapping providers, and credential mapping providers have deployable versions of their Provider SSPIs.

> **Note**
>
> If your security provider database (which stores security policies, security roles, and credentials) is read-only, you can implement the non-deployable version of the SSPI for your authorization, role mapping, and credential mapping security providers. However, you will still need to configure deployable versions of these security provider that do handle deployment.

The DeployableAuthorizationProviderV2 SSPI

An authorization provider that supports deploying security policies on behalf of Web application or Enterprise JavaBean (EJB) deployments needs to implement the `DeployableAuthorizationProviderV2` SSPI instead of the `AuthorizationProvider` SSPI. (However, because the `DeployableAuthorizationProviderV2` SSPI extends the `AuthorizationProvider` SSPI, you actually will need to implement the methods from both SSPIs.) This is because Web application and EJB deployment activities require the authorization provider to perform additional tasks, such as creating and removing security policies. In a security realm, at least one authorization provider must support the `DeployableAuthorizationProviderV2` SSPI, or else it will be impossible to deploy Web applications and EJBs.

> **Note**
>
> For more information about security policies, see Security Policies in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

The DeployableRoleProviderV2 SSPI

A role mapping provider that supports deploying security roles on behalf of Web application or Enterprise JavaBean (EJB) deployments needs to implement the `DeployableRoleProviderV2` SSPI instead of the `RoleProvider` SSPI. (However, because the `DeployableRoleProviderV2` SSPI extends the `RoleProvider` SSPI, you will actually need to implement the methods from both SSPIs.) This is because Web application and EJB deployment activities require the role mapping provider to perform additional tasks, such as creating and removing security roles. In a security realm, at least one role mapping provider must support this SSPI, or else it will be impossible to deploy Web applications and EJBs.

> **Note**
>
> See Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

The DeployableCredentialProvider SSPI

> **Note**
>
> The DeployableCredentialProvider interface is deprecated in this release of WebLogic Server.

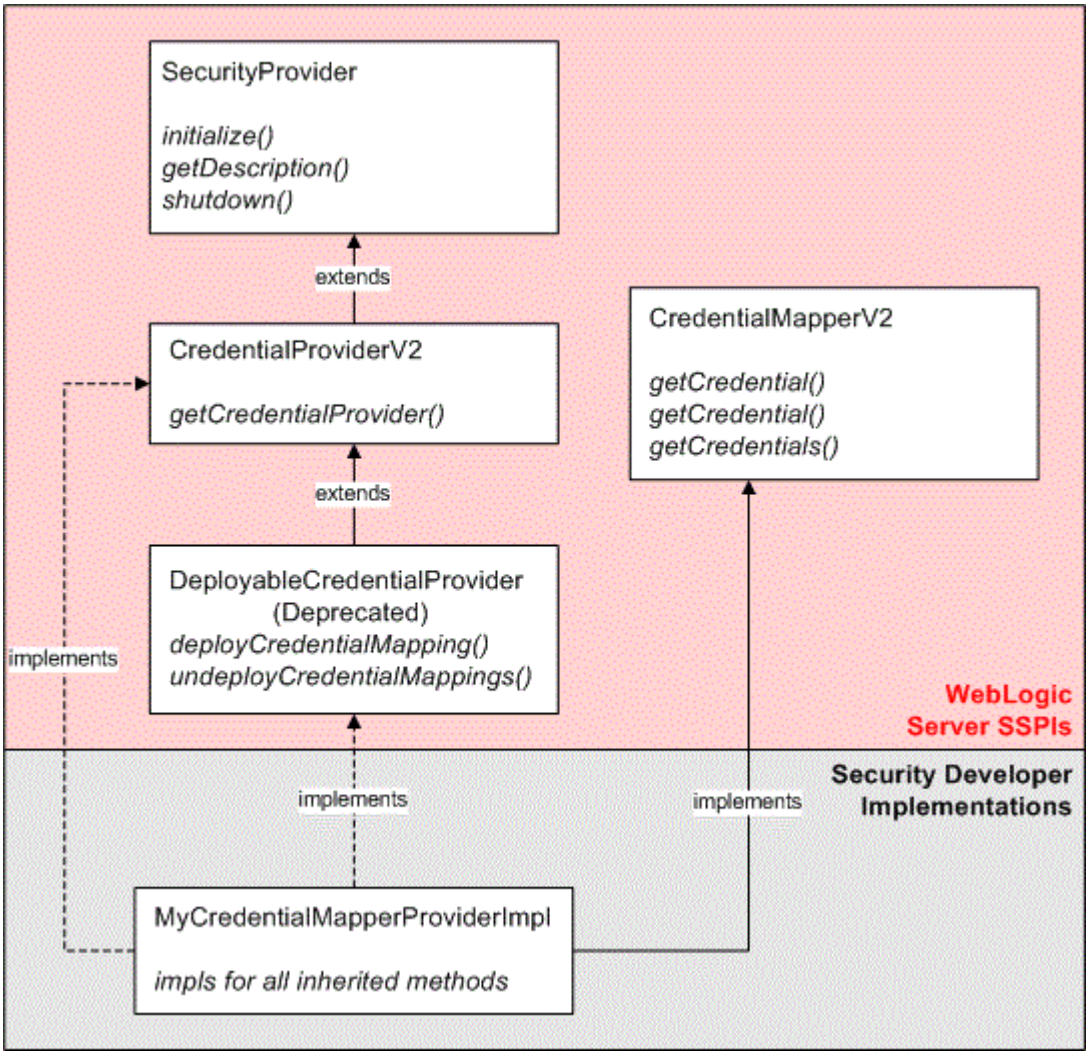
A credential mapping provider that supports deploying security policies on behalf of Resource Adapter (RA) deployments needs to implement the `DeployableCredentialProvider` SSPI instead of the `CredentialProvider` SSPI. (However, because the `DeployableCredentialProvider` SSPI extends the `CredentialProvider` SSPI, you will actually need to implement the methods from both SSPIs.) This is because Resource Adapter deployment activities require the credential mapping provider to perform additional tasks, such as creating and removing credentials and mappings. In a security realm, at least one credential mapping provider must support this SSPI, or else it will be impossible to deploy Resource Adapters.

> **Note**
>
> See Credential Mapping Concepts. See Security Policies in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

Figure 2-3 uses a credential mapping provider to illustrate the inheritance hierarchy that is common to all SSPIs, and shows how a runtime class you supply can implement those interfaces. In this example, Oracle supplies the `SecurityProvider` interface, and the `CredentialProviderV2` and `CredentialMapperV2` SSPIs. Figure 2-3 shows *a single runtime class* called `MyCredentialMapperProviderImpl` that implements the `CredentialProviderV2` *and* `CredentialMapperV2` SSPIs.

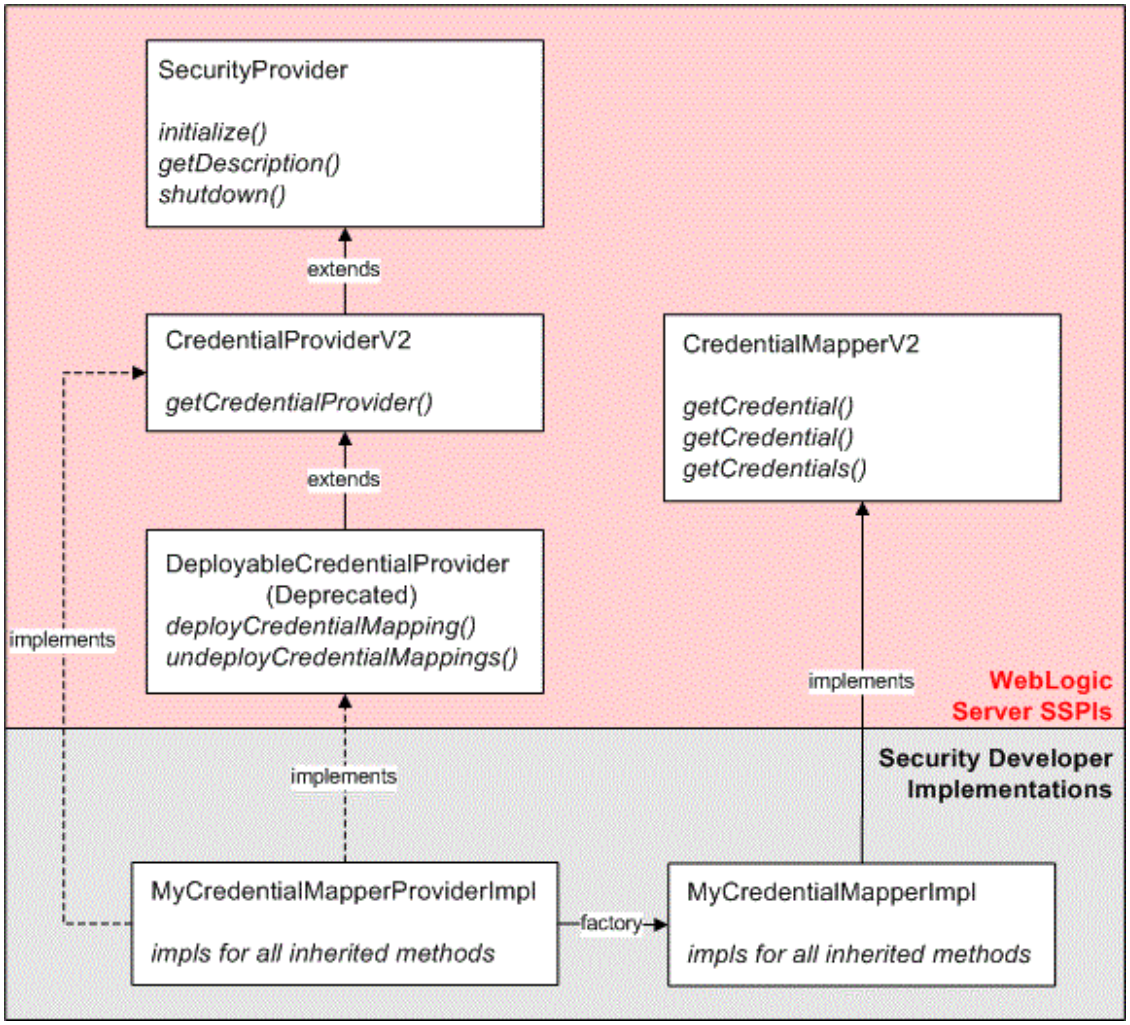
Figure 2-3 Credential Mapping SSPIs and a Single Runtime Class



However, Figure 2-3 illustrates only one way you can implement SSPIs: by creating a *single* runtime class. If you prefer, you can have two runtime classes (as shown in Figure 2-4): one for the implementation of the SSPI ending in `Provider` (for example, `CredentialProviderV2`), and one for the implementation of the other SSPI (for example, the `CredentialMapperV2` SSPI).

When there are separate runtime classes, the class that implements the SSPI ending in `Provider` acts as a factory for generating the runtime class that implements the other SSPI. For example, in Figure 2-4, `MyCredentialMapperProviderImpl` acts as a factory for generating `MyCredentialMapperImpl`.

Figure 2-4 Credential Mapping SSPIs and Two Runtime Classes



> **Note**
>
> If you decide to have two runtime implementation classes, you need to remember to include both runtime implementation classes in the MBean JAR File (MJF) when you generate the security provider's MBean type. See Generating an MBean Type to Configure and Manage the Custom Security Provider .

SSPI Quick Reference

Table 2-1 maps the types of security providers (and their components) with the SSPIs and other interfaces you use to develop them.

Table 2-1 Security Providers, Their Components, and Corresponding SSPIs

Type/Component	SSPIs/Interfaces
Authentication provider	`AuthenticationProviderV2`

Table 2-1 (Cont.) Security Providers, Their Components, and Corresponding SSPIs

Type/Component	SSPIs/Interfaces
LoginModule (JAAS)	`LoginModule`
Identity Assertion provider	`AuthenticationProviderV2`
Identity Asserter	`IdentityAsserterV2`
Principal Validation provider	`PrincipalValidator`
Authorization	`AuthorizationProvider` `DeployableAuthorizationProviderV2`
Access Decision	`AccessDecision`
Adjudication provider	`AdjudicationProviderV2`
Adjudicator	`AdjudicatorV2`
Role Mapping provider	`RoleProvider` `DeployableRoleProviderV2`
Role Mapper	`RoleMapper`
Auditing provider	`AuditProvider`
Audit Channel	`AuditChannel`
Credential Mapping provider	`CredentialProviderV2`
Credential Mapper	`CredentialMapperV2`
Cert Path Provider	`CertPathProvider`
Versionable Application Provider	`VersionableApplicationProvider`

> **Note**
>
> The SSPIs you use to create runtime classes for custom security providers are located in the `weblogic.security.spi.package` in the *Java API Reference for Oracle WebLogic Server*.

Security Service Provider Interface (SSPI) MBeans

As described in Overview of the Development Process, the second step in developing a custom security provider is generating an MBean type for the custom security provider. This section helps you:

- Understand Why You Need an MBean Type
- Determine Which SSPI MBeans to Extend and Implement
- Understand the Basic Elements of an MBean Definition File (MDF)
- Understand the SSPI MBean Hierarchy and How It Affects the WebLogic Remote Console
- Understand What the WebLogic MBeanMaker Provides

Additionally, this section provides SSPI MBean Quick Reference that indicates which required SSPI MBeans must be extended and which optional SSPI MBeans can be implemented for each type of security provider.

Understand Why You Need an MBean Type

In addition to creating runtime classes for a custom security provider, you must also generate an MBean type. The term **MBean** is short for managed bean, a Java object that represents a Java Management eXtensions (JMX) manageable resource.

> **Note**
>
> JMX is a specification that defines a standard management architecture, APIs, and management services. See Understanding JMX in *Developing Manageable Applications Using JMX for Oracle WebLogic Server*.

An **MBean type** is a factory for instances of MBeans. After they are created, you can configure and manage the inherited standard attributes of the custom security provider using the MBean instance, through the WebLogic Remote Console. However, you cannot use the WebLogic Remote Console to manage the custom attributes from your MDF file.

> **Note**
>
> All MBean instances are aware of their parent type, so if you modify the configuration of an MBean type, all instances that you or an administrator may have created will also update their configurations. (For more information, see Understand the SSPI MBean Hierarchy and How It Affects the WebLogic Remote Console .)

Determine Which SSPI MBeans to Extend and Implement

You use MBean interfaces called **SSPI MBeans** to create MBean types. There are two types of SSPI MBeans you can use to create an MBean type for a custom security provider:

- **Required SSPI MBeans**, which you must extend because they define the basic methods that allow a security provider to be configured and managed within the WebLogic Server environment.
- **Optional SSPI MBeans**, which you can implement because they define additional methods for managing security providers. Different types of security providers are able to use different optional SSPI MBeans.

See SSPI MBean Quick Reference.

Understand the Basic Elements of an MBean Definition File (MDF)

An **MBean Definition File (MDF)** is an XML file used by the WebLogic MBeanMaker utility to generate the Java files that comprise an MBean type. All MDFs *must* extend a required SSPI MBean that is specific to the type of the security provider you have created, and *can* implement optional SSPI MBeans.

Example 2-1 shows a sample MBean Definition File (MDF), and an explanation of its content follows. (Specifically, it is the MDF used to generate an MBean type for the WebLogic Credential Mapping provider. Note that the DeployableCredentialProvider interface is deprecated in this release of WebLogic Server.)

> **Note**
>
> A complete reference of MDF element syntax is available in MBean Definition File (MDF) Element Syntax.

Example 2-1 DefaultCredentialMapper.xml

```
<MBeanType
 Name = "DefaultCredentialMapper"
 DisplayName = "DefaultCredentialMapper"
 Package = "weblogic.security.providers.credentials"
 Extends = "weblogic.management.security.credentials. DeployableCredentialMapper"
 Implements = "weblogic.management.security.credentials. UserPasswordCredentialMapEditor,
weblogic.management.security.credentials.UserPasswordCredentialMapExtendedReader,
weblogic.management.security.ApplicationVersioner,
weblogic.management.security.Import,
weblogic.management.security.Export"
PersistPolicy = "OnUpdate"
 Description = "This MBean represents configuration attributes for the WebLogic Credential
Mapping provider.&lt;p&gt;"
>
<MBeanAttribute
 Name = "ProviderClassName"
 Type = "java.lang.String"
 Writeable = "false"
 Default = "&quot;weblogic.security.providers.credentials. DefaultCredentialMapperProviderImpl&quot;"
 Description = "The name of the Java class that loads the WebLogic Credential Mapping
provider."
/>
<MBeanAttribute
 Name = "Description"
 Type = "java.lang.String"
 Writeable = "false"
 Default = "&quot;Provider that performs Default Credential Mapping&quot;"
 Description = "A short description of the WebLogic Credential Mapping provider."
/>
<MBeanAttribute
 Name = "Version"
 Type = "java.lang.String"
 Writeable = "false"
 Default = "&quot;1.0&quot;"
 Description = "The version of the WebLogic Credential Mapping provider."
/>
:
:
</MBeanType>
```

The bold attributes in the `<MBeanType>` tag show that this MDF is named `DefaultCredentialMapper` and that it extends the required SSPI MBean called `DeployableCredentialMapper`. It also includes additional management capabilities by implementing the `UserPasswordCredentialMapEditor` optional SSPI MBean.

The `ProviderClassName`, `Description`, and `Version` attributes defined in the `<MBeanAttribute>` tags are required in any MDF used to generate MBean types for security providers because they define the security provider's basic configuration methods, and are inherited from the base required SSPI MBean called `Provider` (see Figure 2-5). The `ProviderClassName` attribute is especially important. The value for the `ProviderClassName` attribute is the Java filename of the security provider's runtime class (that is, the

implementation of the appropriate SSPI ending in `Provider`). The example runtime class shown in Example 2-1 is `DefaultCredentialMapperProviderImpl.java`.

While not shown in Example 2-1, you can include additional attributes and operations in an MDF using the `<MBeanAttribute>` and `<MBeanOperation>` tags.

> **Note**
>
> The Sample Auditing provider provides an example of adding a custom attribute.

Custom Providers and Classpaths

Classes loaded from `WL_HOME\server\lib\mbeantypes` are not visible to other JAR and EAR files deployed on WebLogic Server. If you have common utility classes that you want to share, you must place them in the system classpath.

> **Note**
>
> `WL_HOME\server\lib\mbeantypes` is the default directory for installing MBean types. Beginning with 9.0, security providers can be loaded from `...\domaindir\lib\mbeantypes` as well. JAR files loaded from the `...\domaindir\lib\mbeantypes` directory can be shared across applications. They do not need to be explicitly placed in the system classpath.

Throwing Exceptions from MBean Operations

Your custom provider MBeans must throw only JDK exception types or `weblogic.management.utils` exception types. Otherwise, JMX clients may not include the code necessary to receive your exceptions.

- For typed exceptions, you must throw only the exact types from the throw clause of your MBean's method, as opposed to deriving and throwing your own exception type from that type.
- For nested exceptions, you must throw only JDK exception types or `weblogic.management.utils` exceptions.
- For runtime exceptions, you must throw or pass through only JDK exceptions.

Specifying Non-Clear Text Values for MBean Attributes

As described in The MBeanAttribute Subelement, you can use the Encrypted attribute to specify that the value of an MBean attribute should not be displayed as clear text. For example, you encrypt the value of the MBean attribute when getting input for a password. The following code fragment shows an example of using the Encrypted attribute:

```
<MBeanAttribute
Name        = "PrivatePassPhrase"
Type        = "java.lang.String"
Encrypted   = "true"
Default     = "&amp;quot;&amp;quot;"
Description = "The Keystore password."
/>
```

Using Dynamic MBean Attributes

A *Dynamic MBean* is an MBean that defines its management interface at run-time. For example, a configuration MBean could determine the names and types of the attributes it exposes by parsing an XML file. MBean attributes marked as dynamic can be changed without requiring a server restart. If you declare an MBean attribute to be dynamic, then its value is automatically updated as it is changed in the running server. When the security provider implementation reads the value from the MBean object passed at runtime to the `initialize` method, the provider receives the most recently updated value.

For more information about:

- The `initialize` method of the security provider interface, see Understand the Purpose of the Provider SSPIs.
- Dynamic MBean attributes and how to specify them, see The MBeanAttribute Subelement.

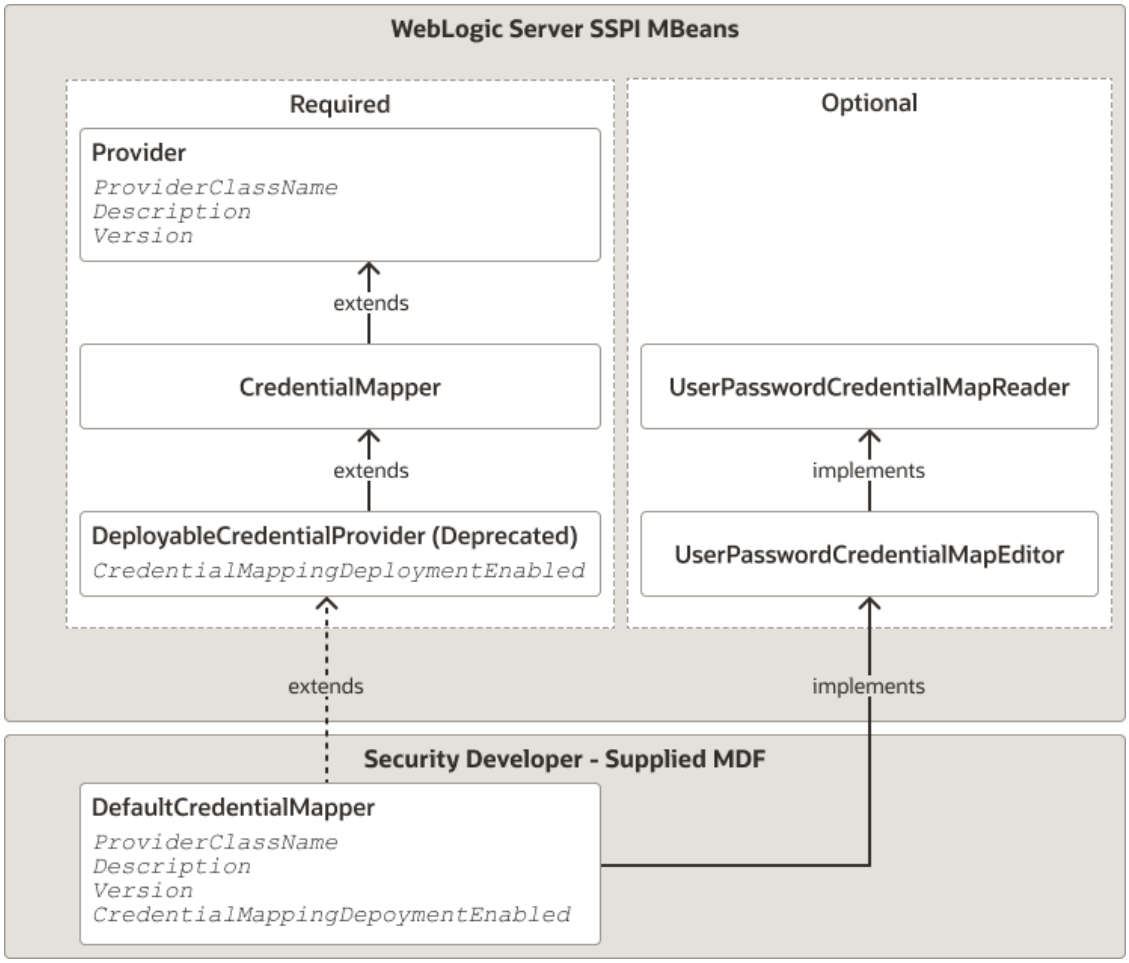
Understand the SSPI MBean Hierarchy and How It Affects the WebLogic Remote Console

The inherited standard attributes that are specified in the required SSPI MBeans that your MBean Definition File (MDF) extends (all the way up to the Provider base SSPI MBean) automatically appear in the WebLogic Remote Console page for the associated security provider. You can use these attributes to configure and manage your custom security providers.

> **Note**
>
> For authentication security providers only, the attributes and operations that are specified in the optional SSPI MBeans your MDF implements are also automatically supported by the WebLogic Remote Console. For other types of security providers, you must write a console extension in order to make the attributes and operations inherited from the optional SSPI MBeans available in the WebLogic Remote Console.

Figure 2-5 illustrates the SSPI MBean hierarchy for security providers (using the WebLogic Credential Mapping MDF as an example), and indicates what attributes and operations will appear in the WebLogic Remote Console for the WebLogic Credential Mapping provider.

Figure 2-5 SSPI MBean Hierarchy for Credential Mapping Providers



Implementing the hierarchy of SSPI MBeans in the `DefaultCredentialMapper` MDF generates a page in the WebLogic Remote Console that lists the standard inherited properties of the custom provider.

The Name, Description, and Version fields come from attributes with these names inherited from the base required SSPI MBean called `Provider` and specified in the `DefaultCredentialMapper` MDF. Note that the `DisplayName` attribute in the `DefaultCredentialMapper` MDF generates the value for the Name field, and that the `Description` and `Version` attributes generate the values for their respective fields as well. The Credential Mapping Deployment Enabled field is displayed (on the Provider Specific page) because of the `CredentialMappingDeploymentEnabled` attribute in the `DeployableCredentialMapper` required SSPI MBean, which the `DefaultCredentialMapper` MDF extends.

Understand What the WebLogic MBeanMaker Provides

The **WebLogic MBeanMaker** is a command-line utility that takes an MBean Definition File (MDF) as input and outputs files for an MBean type. When you run the MDF you created through the WebLogic MBeanMaker, the following occurs:

- Any attributes inherited from required SSPI MBeans—as well as any custom attributes you added to the MDF—cause the WebLogic MBeanMaker to generate *complete getter/setter*

methods in the MBean type's information file. (The MBean information file is not shown in Figure 2-6.) See About the MBean Information File.

Necessary developer action: None. No further work must be done for these methods.

- Any operations inherited from optional SSPI MBeans cause the MBean implementation file to inherit their methods, whose implementations you must supply from scratch.

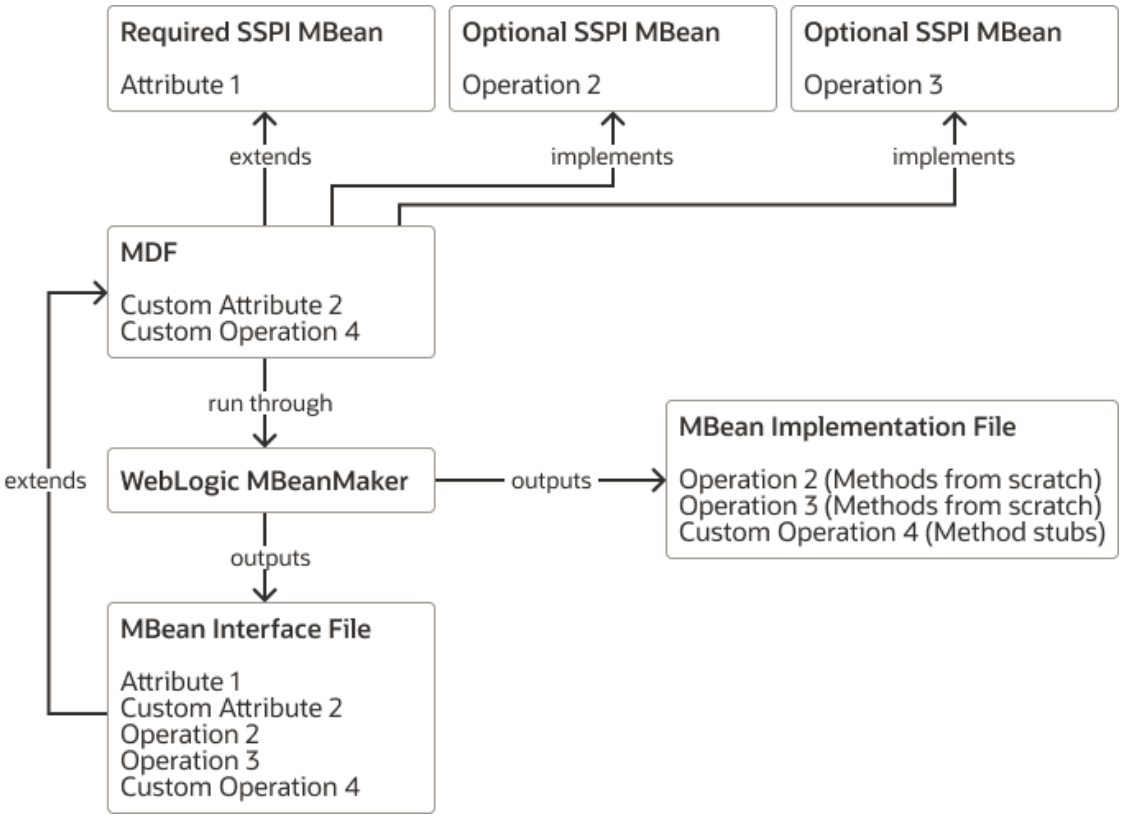
  Necessary developer action: Currently, the WebLogic MBeanMaker does not generate method stubs for these inherited methods, so you will need to supply the appropriate implementations.

- Any custom operations you added to the MDF will cause the WebLogic MBeanMaker to *generate method stubs*.

  Necessary developer action: You must provide implementations for these methods. (However, because the WebLogic MBeanMaker generates the stubs, you do not need to look up the Java method signatures.)

This is illustrated in Figure 2-6.

Figure 2-6 What the WebLogic MBeanMaker Provides



About the MBean Information File

The MBean information file contains a compiled definition of the data in the MBean Definition File in a form that JMX Model MBeans require. The format of this file is a list of attributes, operations, and notifications, each of which also has a set of descriptor tags that describe that entity. In addition, the MBean itself also has a set of descriptor tags. An example of this format is as follows:

```
MBean + tags
attribute1 + tags, attribute2 + tags ...
operation1 + tags, operation2 + tags ...
notification1 + tags, notification2 + tags ...
```

If desired, you can access this information at runtime by calling the standard JMX server `getMBeanInfo` method to obtain the `ModelMBeanInfo`.

> **Note**
>
> Be sure to reference the JMX specification to determine how to interpret the returned structure.

SSPI MBean Quick Reference

Based on the list of SSPIs you need to implement as part of developing your custom security provider, locate the required SSPI MBeans you need to extend in Table 2-2. Using Table 2-3 through Table 2-5, locate any optional SSPI MBeans you also want to implement for managing your security provider.

Table 2-2 Required SSPI MBeans

Type	Package Name	Required SSPI MBean
Authentication provider	`authentication`	`Authenticator`
Identity Assertion provider	`authentication`	`IdentityAsserter`
Authorization provider	`authorization`	`Authorizer` or `DeployableAuthorizer`
Adjudication provider	`authorization`	`Adjudicator`
Role Mapping provider	`authorization`	`RoleMapper` or `DeployableRoleMapper`
Auditing provider	`audit`	`Auditor`
Credential Mapping provider	`credentials`	`CredentialMapper` or `DeployableCredentialMapper`
Cert Path Provider	`pk`	`CertPathBuilder` or `CertPathValidator`

> **Note**
>
> The required SSPI MBeans shown in Table 2-2 are located in the `weblogic.management.security.<Package_Name>` package.

Table 2-3 Optional Authentication SSPI MBeans

Optional SSPI MBeans	Purpose
`GroupEditor`	Create a group. If the group already exists, an exception is thrown.

Table 2-3 (Cont.) Optional Authentication SSPI MBeans

Optional SSPI MBeans	Purpose
`GroupMemberLister`	List a group's members.
`GroupReader`	Read data about groups.
`GroupRemover`	Remove groups.
`MemberGroupLister`	List the groups containing a user or a group.
`UserEditor`	Create, edit and remove users.
`UserPasswordEditor`	Change a user's password.
`UserReader`	Read data about users.
`UserRemover`	Remove users.

> **Note**
>
> The optional authentication SSPI MBeans shown in Table 2-3 are located in the `weblogic.management.security.authentication` package.
>
> For an example of how to implement the optional authentication SSPI MBeans shown in Table 2-4, review the code for the Manageable Sample Authentication Provider.

Table 2-4 Optional Authorization SSPI MBeans

Optional SSPI MBeans	Purpose
`PolicyAuxiliary`	Auxiliary methods for creating, editing, and removing policies.
`PolicyConsumer`	Indicates that the provider supports policy consumption.
`PolicyEditor`	Create, edit and remove security policies.
`PolicyLister`	List data about policies.
`PolicyReader`	Read data about security policies.
`PolicyStore`	Manages policies in a policy store.
`RoleEditor`	Create, edit and remove security roles.
`RoleReader`	Read data about security roles.
`RoleLister`	List data about roles.

> **Note**
>
> The optional authorization SSPI MBeans shown in Table 2-4 are located in the `weblogic.management.security.authorization` package.

Table 2-5 Optional Credential Mapping SSPI MBeans

Optional SSPI MBeans	Purpose
`UserPasswordCredentialMapEditor`	Edit credential maps that map a WebLogic user to a remote username and password.
`UserPasswordCredentialMapExtendedReader`	Read credential maps that map a WebLogic user to a remote username and password.
`UserPasswordCredentialMapReader`	Read credential maps that map a WebLogic user to a remote username and password.

> **Note**
>
> The optional credential mapping SSPI MBeans shown in Table 2-5 are located in the `weblogic.management.security.credentials` package.

Security Data Migration

Several of the WebLogic security providers have been developed to support security data migration. This means that administrators can export users and groups (for the WebLogic Authentication provider), security policies (for the WebLogic Authorization provider), security roles (for the WebLogic Role Mapping provider), or credential mappings (for the credential mapping provider) from one security realm, and then import them into another security realm. Administrators can migrate security data for each of these WebLogic security providers individually, or migrate security data for all the WebLogic security providers at once (that is, security data for the entire security realm).

The migration of security data may be helpful to administrators when:

- Transitioning from development mode to production mode
- Proliferating production mode security configurations to security realms in new WebLogic Server domains
- Moving data to a new security realm in the same WebLogic Server domain or in a different WebLogic Server domain.
- Moving from one security realm to a new security realm in the same WebLogic Server domain, where one or more of the WebLogic security providers will be replaced with custom security providers. (In this case, administrators need to copy security data for the security providers that are not being replaced.)

The following sections provide more information about security data migration:

- Migration Concepts
- Adding Migration Support to Your Custom Security Providers

Migration Concepts

Before you start to work with security data migration, you need to understand the following concepts:

- Formats

- Constraints
- Migration Files

Formats

A **format** is simply a data format that specifies how security data should be exported or imported. Currently, WebLogic Server does not provide any standard, public formats for developers of security providers. Therefore, the format you use is entirely up to you. Keep in mind, however, that for data to be exported from one security provider and later imported to another security provider, both security providers must understand how to process the same format. **Supported formats** are the list of data formats that a given security provider understands how to process.

> **Note**
>
> Because the data format used for the WebLogic security providers is unpublished, you cannot currently migrate security data from a WebLogic security provider to a custom security provider, or visa versa.
>
> Additionally, security vendors wanting to exchange security data with security providers from other vendors will need to collaborate on a standard format to do so.

Constraints

Constraints are key/value pairs used to specify options to the export or import process. Constraints allow administrators to control which security data is exported or imported from the security provider's database. For example, an administrator may want to export only users (not groups) from an authentication provider's database, or a subset of those users. **Supported constraints** are the list of constraints that administrators *may* specify during the migration process for a particular security provider. For example, an authentication provider's database can be used to import users and groups, but not security policies.

Migration Files

Export files are the files to which security data is written (in the specified format) during the export portion of the migration process. **Import files** are the files from which security data is read (also in the specified format) during the import portion of the migration process. Both export and import files are simply temporary storage locations for security data as it is migrated from one security provider's database to another.

> **Note**
>
> The migration files are not protected unless you take additional measures to protect them. Because migration files may contain sensitive data, take extra care when working with them.

Adding Migration Support to Your Custom Security Providers

If you want to develop custom security providers that support security data migration like the WebLogic security providers do, you need to extend the

`weblogic.management.security.ImportMBean` and `weblogic.management.security.ExportMBean` optional SSPI MBeans in the MBean Definition File (MDF) that you use to generate MBean types for your custom security providers, then implement their methods. These optional SSPI MBeans include the attributes and operations described in Table 2-6 and Table 2-7, respectively.

Table 2-6 Attributes and Operations of the ExportMBean Optional SSPI MBean

Attributes/Operations	Description
`SupportedExportFormats`	A list of export data formats that the security provider supports.
`SupportedExportConstraints`	A list of export constraints that the security provider supports.
`exportData`	Exports provider-specific security data in a specified format.
`format`	A parameter on the `exportData` operation that specifies the format to use for exporting provider-specific data.
`filename`	A parameter on the `exportData` operation that specifies the full path to the filename used to export provider-specific data. **Notes:** The WebLogic security providers that support security data migration are implemented in a way that allows you to specify a relative path (from the directory relative to the server you are working on). You must specify a directory that already exists; WebLogic Server will *not* create one for you.
`constraints`	A parameter on the `exportData` operation that specifies the constraints to be used when exporting provider-specific data.

> **Note**
>
> See `ExportMBean interface` in *Java API Reference for Oracle WebLogic Server*.

Table 2-7 Attributes and Operations of the ImportMBean Optional SSPI MBean

Attributes/Operations	Description
`SupportedImportFormats`	A list of import data formats that the security provider supports.
`SupportedImportConstraints`	A list of import constraints that the security provider supports.
`importData`	Imports provider-specific data from a specified format.
`format`	A parameter on the `importData` operation that specifies the format to use for importing provider-specific data.

Table 2-7 (Cont.) Attributes and Operations of the ImportMBean Optional SSPI MBean

Attributes/Operations	Description
`filename`	A parameter on the `importData` operation that specifies the full path to the filename used to import provider-specific data. **Note:** The WebLogic security providers that support security data migration are implemented in a way that allows you to specify a relative path (from the directory relative to the server you are working on). You must specify a directory that already exists; WebLogic Server will *not* create one for you.
`constraints`	A parameter on the `importData` operation that specifies the constraints to be used when importing provider-specific data.

> **Note**
>
> See `ImportMBean interface` in *Java API Reference for Oracle WebLogic Server*.

Management Utilities Available to Developers of Security Providers

The `weblogic.management.utils` package contains additional management interfaces and exceptions that developers might find useful, particularly when generating MBean types for their custom security providers. Implementation of these interfaces and exceptions is not required to develop a custom security provider (unless you inherit them by implementing optional SSPI MBeans in your custom security provider's MDF).

> **Note**
>
> The interfaces and classes are located in this package (rather than in weblogic.management.security) because they are general purpose utilities; in other words, these utilities can also be used for non-security MBeans. The various types of MBeans are described in Overview of WebLogic Server Subsystem MBeans in *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*.

The `weblogic.management.utils` package contains the following utilities:

- Common exceptions.
- Interfaces that provide methods for handling large lists of data.
- An interface containing configuration attributes that are required to communicate with an external LDAP server.

> **Note**
>
> The Manageable Sample Authentication provider uses the `weblogic.management.utils` package for exceptions as well as to handle lists of data.

See *Java API Reference for Oracle WebLogic Server* for the `weblogic.management.utils` package.

Security Providers and WebLogic Resources

A **WebLogic resource** is a structured object used to represent an underlying WebLogic Server entity that can be protected from unauthorized access. Developers of custom authorization, role mapping, and credential mapping providers need to understand how these security providers interact with WebLogic resources and the security policies used to secure those resources.

> **Note**
>
> Security policies replace the access control lists (ACLs) and permissions that were used to protect WebLogic resources in previous releases of WebLogic Server.

The following sections provide information about security providers and WebLogic resources:

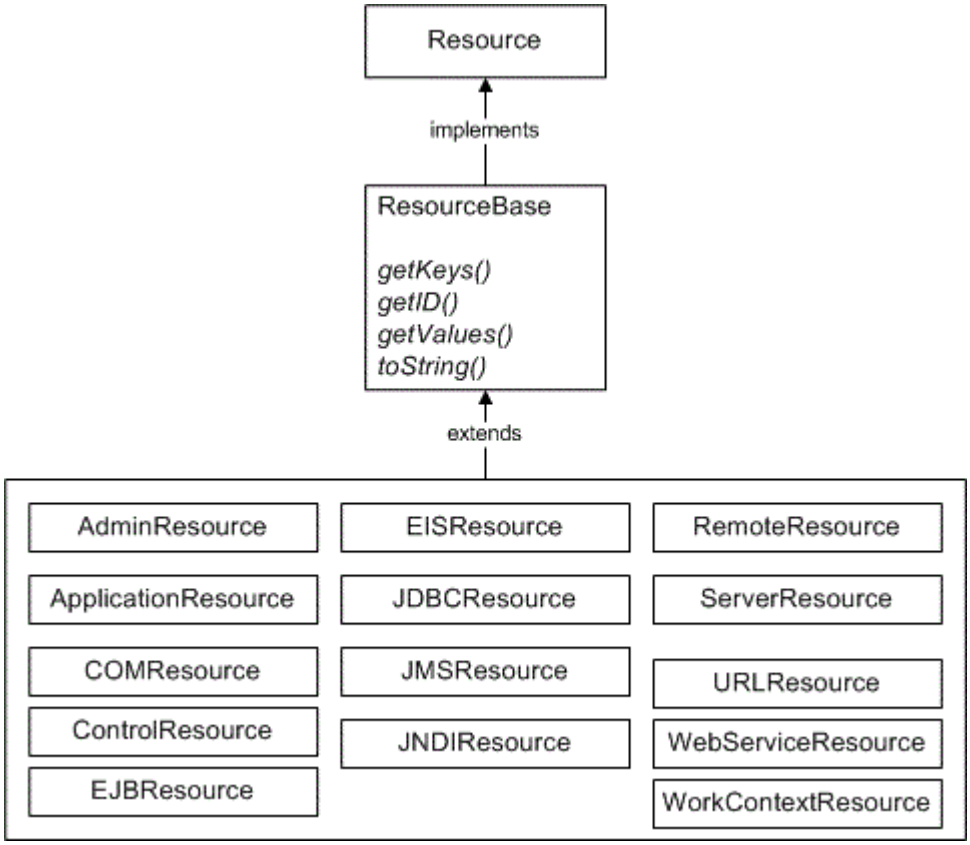
- The Architecture of WebLogic Resources
- Types of WebLogic Resources
- WebLogic Resource Identifiers
- Creating Default Groups for WebLogic Resources
- Creating Default Security Roles for WebLogic Resources
- Creating Default Security Policies for WebLogic Resources
- Single-Parent Resource Hierarchies
- ContextHandlers and WebLogic Resources

> **Note**
>
> See *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

The Architecture of WebLogic Resources

The `Resource` interface, located in the `weblogic.security.spi` package, provides the definition for an object that represents a WebLogic resource, which can be protected from unauthorized access. The `ResourceBase` class, located in the `weblogic.security.service` package, is an abstract base class for more specific WebLogic resource types, and facilitates the model for extending resources. (See Figure 2-7 and Types of WebLogic Resources .)

Figure 2-7 Architecture of WebLogic Resources



The `ResourceBase` class includes the Oracle-provided implementations of the `getID`, `getKeys`, `getValues`, and `toString` methods. See *WebLogic Server API Reference Javadoc* for the `ResourceBase` class.

This architecture allows you to develop security providers without requiring that they be aware of any particular WebLogic resources. Therefore, when new resource types are added, you should not need to modify the security providers.

Types of WebLogic Resources

As shown in Figure 2-7, certain classes in the `weblogic.security.service` package extend the `ResourceBase` class, and therefore provide you with implementations for specific types of WebLogic resources. WebLogic resource implementations are available for:

- Administrative resources
- Application resources
- COM resources
- Control resources
- EIS resources
- EJB resources
- JDBC resources
- JMS resources

- JNDI resources
- Remote resources
- Server resources
- URL resources
- Web service resources
- Work Context resources

> **Note**
>
> See *Securing Resources Using Roles and Policies for Oracle WebLogic Server* and the *Java API Reference for Oracle WebLogic Server* for the `weblogic.security.service` package.

WebLogic Resource Identifiers

Each WebLogic resource (described in Types of WebLogic Resources) can be identified in two ways: by its `toString()` representation or by an ID obtained using the `getID()` method.

The toString() Method

If you use the `toString()` method of any WebLogic resource implementation, a description of the WebLogic resource will be returned in the form of a `String`. First, the type of the WebLogic resource is printed in pointy-brackets. Then, each key is printed, in order, along with its value. The keys are comma-separated. Values that are lists are comma-separated and delineated by open and close curly braces. Each value is printed as is, except that commas (,), open braces ({), close braces (}), and back slashes (\) are each escaped with a back slash. For example, the EJB resource:

```
EJBResource ('myApp",
             'MyJarFile",
             'myEJB",
             'myMethod",
             'Home",
             new String[] {'argumentType1", 'argumentType2"}
            );
```

will produce the following `toString` output:

```
type=<ejb>, app=myApp, module="MyJarFile", ejb=myEJB, method="myMethod",
 methodInterface="Home", methodParams={argumentType1, argumentType2}
```

The format of the WebLogic resource description provided by the `toString()` method is public (that is, you can construct one without using a `Resource` object) and is reversible (meaning that you can convert the `String` form back to the original WebLogic resource).

> **Note**
>
> Example 2-2 illustrates how to use the `toString()` method to identify a WebLogic resource.

Resource IDs and the getID() Method

The `getID()` method on each of the defined WebLogic resource types returns a 64-bit hashcode that can be used to uniquely identify the WebLogic resource in a security provider. The resource ID can be effectively used for fast runtime caching, using the following algorithm:

1. Obtain a WebLogic resource.
2. Get the resource ID for the WebLogic resource using the `getID` method.
3. Look up the resource ID in the cache.
4. If the resource ID is found, then return the security policy.
5. If the resource ID is not found, then:
   a. Use the `toString()` method to look up the WebLogic resource in the security provider database.
   b. Store the resource ID and the security policy in cache.
   c. Return the security policy.

> **Note**
>
> Example 2-3 illustrates how to use the `getID()` method to identify a WebLogic resource in an authorization provider, and provides a sample implementation of this algorithm.

Because it is not guaranteed stable across multiple runs, you should not use the resource ID to store information about the WebLogic resource in a security provider database. Instead, Oracle recommends that you store any resource-to-security policy and resource-to-security role mappings in their corresponding security provider database using the WebLogic resource's `toString()` method.

> **Note**
>
> See Initialization of the Security Provider Databaseand The toString() Method.

Creating Default Groups for WebLogic Resources

When writing a runtime class for a custom authentication provider, there are several default groups that you are required to create. Table 2-8 provides information to assist you with this task.

Table 2-8 Default Groups and Group Membership

Group Name	Group Membership
`Administrators`	Empty, or an administrative user.
`Deployers`	Empty
`Monitors`	Empty
`Operators`	Empty

Table 2-8 (Cont.) Default Groups and Group Membership

Group Name	Group Membership
`AppTesters`	Empty
`OracleSystemGroup`	`OracleSystemUser`

Creating Default Security Roles for WebLogic Resources

When writing a runtime class for a custom role mapping provider, there are several default global roles that you are required to create. Table 2-9 provides information to assist you with this task.

Table 2-9 Default Global Roles and Group Associations

Global Role Name	Group Association
`Admin`	`Administrators` group
`AdminChannelUser`	`AdminChannelUsers`, `Administrators`, `Deployers`, `Operators`, `Monitors`, and `AppTesters` groups
`Anonymous`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`CrossDomainConnector`	`CrossDomainConnectors` group
`Deployer`	`Deployers` group
`Monitor`	`Monitors` group
`Operator`	`Operators` group
`AppTester`	`AppTesters` group
`OracleSystemRole`	`OracleSystemGroup`

> **Note**
>
> See Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Creating Default Security Policies for WebLogic Resources

When writing a runtime class for a custom authorization provider, there are several default security policies that you are required to create. These default security policies initially protect the various types of WebLogic resources. Table 2-10 provides information to assist you with this task.

Table 2-10 Default Security Policies for WebLogic Resources

WebLogic Resource Constructor	Security Policy
`new AdminResource(null, null, null)`	`Admin` global role

Table 2-10 (Cont.) Default Security Policies for WebLogic Resources

WebLogic Resource Constructor	Security Policy
`new AdminResource("Configuration", null, null)`	`Admin`, `Deployer`, `Monitor`, or `Operator` global roles
`new AdminResource('FileDownload", null, null)`	`Admin` or `Deployer` global role
`new AdminResource('FileUpload", null, null)`	`Admin` or `Deployer` global role
`New AdminResource('ViewLog", null, null)`	`Admin` or `Deployer` global role
`new ControlResource(null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`new EISResource(null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`new EJBResource(null, null, null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`new JDBCResource(null, null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`new JNDIResource(null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`new JMSResource(null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`new ServerResource(null, null, null)`	`Admin` or `Operator` global roles
`new URLResource(null, null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`new WebServiceResource(null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group
`new WorkContext(null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` group

> **Note**
>
> Application and COM resources should not have default security policies (that is, they should not grant permission to anyone by default).

Looking Up WebLogic Resources in a Security Provider's Runtime Class

Example 2-2 illustrates how to look up a WebLogic resource in the runtime class of an authorization provider. This algorithm assumes that the security provider database for the authorization provider contains a mapping of WebLogic resources to security policies. It is not required that you use the algorithm shown in Example 2-2, or that you utilize the call to the `getParentResource` method. (See Single-Parent Resource Hierarchies.)

Example 2-2 How to Look Up a WebLogic Resource in an Authorization Provider: Using the toString Method

```
Policy findPolicy(Resource resource) {
   Resource myResource = resource;
   while (myResource != null) {
      String resourceText = myResource.toString();
      Policy policy = lookupInDB(resourceText);
      if (policy != null) return policy;
      myResource = myResource.getParentResource();
   }
   return null;
}
```

You can optimize the algorithm for looking up a WebLogic resource by using the `getID` method for the resource. (Use of the `toString` method alone, as shown in Example 2-2, may impact performance due to the frequency of string concatenations.) The `getID` method may be quicker and more efficient because it is a hash operation that is calculated and cached within the WebLogic resource itself. Therefore, when the `getID` method is used, the `toString` value only needs to be calculated once per resource (as shown in Example 2-3).

Example 2-3 How to Look Up a WebLogic Resource in an Authorization Provider: Using the getID Method

```
Policy findPolicy(Resource resource) {
   Resource myResource = resource;
   while (myResource != null) {
      long id = myResource.getID();
      Policy policy = lookupInCache(id);
      if (policy != null) return policy;
      String resourceText = myResource.toString();
      Policy policy = lookupInDB(resourceText);
      if (policy != null) {
         addToCache(id, policy);
         return policy;
      }
      myResource = myResource.getParentResource();
   }
   return null;
}
```

> **Note**
>
> The `getID` method is not guaranteed between patch sets or future WebLogic Server releases. Therefore, you should not store `getID` values in your security provider database.

Single-Parent Resource Hierarchies

The level of granularity for WebLogic resources is up to you. For example, you can consider an entire Web application, a particular Enterprise JavaBean (EJB) within that Web application, or a single method within that EJB to be a WebLogic resource.

WebLogic resources are arranged in a hierarchical structure ranging from most specific to least specific. You can use the `getParentResource` method for each of the WebLogic resource types if you like, but it is not required.

The WebLogic security providers use the single-parent resource hierarchy as follows: If a WebLogic security provider attempts to access a specific WebLogic resource and that resource cannot be located, the WebLogic security provider will call the `getParentResource` method of that resource. The parent of the current WebLogic resource is returned, and allows the WebLogic security provider to move up the resource hierarchy to protect the next (less-specific) resource. For example, if a caller attempts to access the following URL resource:

```
type=<url>, application=myApp, contextPath="/mywebapp", uri=foo/bar/my.jsp
```

and that exact URL resource cannot be located, the WebLogic security provider will progressively attempt to locate and protect the following resources (in order):

```
type=<url>, application=myApp, contextPath="/mywebapp", uri=/foo/bar/*
type=<url>, application=myApp, contextPath="/mywebapp", uri=/foo/*
type=<url>, application=myApp, contextPath="/mywebapp", uri=*.jsp
type=<url>, application=myApp, contextPath="/mywebapp", uri=/*
type=<url>, application=myApp, contextPath="/mywebapp"
type=<url>, application=myApp
type=<app>, application=myApp
type=<url>
```

> **Note**
>
> See *Java API Reference for Oracle WebLogic Server* for any of the predefined WebLogic resource types or the Resource interface.

Pattern Matching for URL Resources

Sections SRV.11.1 and SRV.11.2 of the Java Servlet 2.3 Specification (http://jcp.org/aboutJava/communityprocess/first/jsr053/index.html) describe the servlet container's pattern matching rules. These rules are used for URL resources as well. The following examples illustrate some important concepts with regard to URL resource pattern matching.

Example 1

For the URL resource `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp, httpMethod=GET`, the resource hierarchy used is as follows. (Note lines 3 and 4, which contain URL patterns that may be different from what is expected.)

1. type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp, httpMethod=GET
2. type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp
3. type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp/*, httpMethod=GET
4. type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp/*
5. type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/*, httpMethod=GET
6. type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/*
7. type=<url>, application=myApp, contextPath=/mywebapp, uri=*.jsp, httpMethod=GET
8. type=<url>, application=myApp, contextPath=/mywebapp, uri=*.jsp
9. type=<url>, application=myApp, contextPath=/mywebapp, uri=/*, httpMethod=GET
10. type=<url>, application=myApp, contextPath=/mywebapp, uri=/*

11. type=<url>, application=myApp, contextPath=/mywebapptype=<url>, application=myApp
12. type=<app>, application=myApp
13. type=<url>

Example 2

For the URL resource `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo`, the resource hierarchy used is as follows. (Note line 2, which contains a URL pattern that may be different from what is expected.)

1. type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo
2. type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/*
3. type=<url>, application=myApp, contextPath=/mywebapp, uri=/*
4. type=<url>, application=myApp, contextPath=/mywebapp
5. type=<url>, application=myApp
6. type=<app>, application=myApp
7. type=<url>

ContextHandlers and WebLogic Resources

A **`ContextHandler`** is a high-performing WebLogic class that obtains additional context and container-specific information from the resource container, and provides that information to security providers making access or role mapping decisions. The `ContextHandler` interface provides a way for an internal WebLogic resource container to pass additional information to a WebLogic Security Framework call, so that a security provider can obtain contextual information beyond what is provided by the arguments to a particular method. A `ContextHandler` is essentially a name/value list and as such, it requires that a security provider know what names to look for. (In other words, use of a `ContextHandler` requires close cooperation between the WebLogic resource container and the security provider.) Each name/value pair in a `ContextHandler` is known as a **context element**, and is represented by a `ContextElement` object.

> **Note**
>
> See *Java API Reference for Oracle WebLogic Server* for the `weblogic.security.service` package.

Resource types have different context elements whose values you can inspect as part of developing a custom provider. That is, not all containers pass all context elements.

Table 2-11 lists the available `ContextHandler` entries.

Table 2-11 Context Handler Entries

Context Element Name	Description and Type
`com.bea.contextelement.servlet.HttpServletRequest`	A servlet access request or SOAP message via HTTP `jakarta.http.servlet.HttpServletRequest`

Table 2-11 (Cont.) Context Handler Entries

Context Element Name	Description and Type
`com.bea.contextelement.servlet.HttpServletResponse`	A servlet access response or SOAP message via HTTP `jakarta.http.servlet.HttpServletResponse`
`com.bea.contextelement.wli.Message`	A WebLogic Integration message. The message is streamed to the audit log. `java.io.InputStream`
`com.bea.contextelement.channel.Port`	The internal listen port of the network channel accepting or processing the request `java.lang.Integer`
`com.bea.contextelement.channel.PublicPort`	The external listen port of the network channel accepting or processing the request `java.lang.Integer`
`com.bea.contextelement.channel.RemotePort`	The port of the remote end of the TCP/IP connection of the network channel accepting or processing the request `java.lang.Integer`
`com.bea.contextelement.channel.Protocol`	The protocol used to make the request of the network channel accepting or processing the request `java.lang.String`
`com.bea.contextelement.channel.Address`	The internal listen address of the network channel accepting or processing the request `java.lang.String`
`com.bea.contextelement.channel.PublicAddress`	The external listen address of the network channel accepting or processing the request `java.lang.String`
`com.bea.contextelement.channel.RemoteAddress`	The remote address of the TCP/IP connection of the network channel accepting or processing the request `java.lang.String`
`com.bea.contextelement.channel.ChannelName`	The name of the network channel accepting or processing the request `java.lang.String`
`com.bea.contextelement.channel.Secure`	Is the network channel accepting or processing the request using SSL? `java.lang.Boolean`
`com.bea.contextelement.ejb20.Parameter[1-N]`	Object based on parameter
`com.bea.contextelement.wsee.SOAPMessage`	`jakarta.xml.rpc.handler.MessageContext`
`com.bea.contextelement.entitlement.EAuxiliaryID`	Used by WebLogic Server internal process. `weblogic.entitlement.expression.EAuxiliary`

Table 2-11 (Cont.) Context Handler Entries

Context Element Name	Description and Type
`com.bea.contextelement.security.ChainPrevalidatedBySSL`	The SSL framework has validated the certificate chain, meaning that the certificates in the chain have signed each other properly; the chain terminates in a certificate that is one of the server's trusted CAs; the chain honors the basic constraints rules; and the certificates in the chain have not expired. `java.lang.Boolean`
`com.bea.contextelement.xml.SecurityToken`	Not used in this release of WebLogic Server. `weblogic.xml.crypto.wss.provider.SecurityToken`
`com.bea.contextelement.xml.SecurityTokenAssertion`	Not used in this release of WebLogic Server. `java.util.Map`
`com.bea.contextelement.webservice.Integrity{id:XXXXX}`	`jakarta.security.auth.Subject`
`com.bea.contextelement.saml.SSLClientCertificateChain`	The SSL client certificate chain obtained from the SSL connection over which a sender-vouches SAML assertion was received. `java.security.cert.X509Certificate[]`
`com.bea.contextelement.saml.MessageSignerCertificate`	The certificate used to sign a Web service message. `java.security.cert.X509Certificate`
`com.bea.contextelement.saml.subject.ConfirmationMethod`	The type of SAML assertion: bearer, artifact, sender-vouches, or holder-of-key. `java.lang.String`
`com.bea.contextelement.saml.subject.dom.KeyInfo`	The `<ds:KeyInfo>` element to be used for subject confirmation with holder-of-key SAML assertions. `org.w3c.dom.Element`

Example 2-4 illustrates how you can access `HttpServletRequest` and `HttpServletResponse` context element objects via a URL (Web) resource's `ContextHandler`. For example, you might use this code in the `isAccessAllowed()` method of your `AccessDecision` SSPI implementation. See Implement the AccessDecision SSPI.)

Example 2-4 Example: Accessing Context Elements in the URL Resource ContextHandler

```
static final String SERVLETREQUESTNAME =
'com.bea.contextelement.servlet.HttpServletRequest";
if (resource instanceof URLResource) {
   HttpServletRequest req =
      (HttpServletRequest)handler.getValue(SERVLETREQUESTNAME);
}
```

> **Note**
>
> You might also want to access these context elements in the `getRoles()` method of the RoleMapper SSPI implementation or the `getContext()` method of the `AuditContext` interface implementation. See Implement the RoleMapper SSPI and Audit Context, respectively.)

Providers and Interfaces that Support Context Handlers

The `ContextHandler` interface provides a way to pass additional information to a WebLogic Security Framework call, so that a security provider or interface can obtain additional context information beyond what is provided by the arguments to a particular method.

Table 2-12 describes the context handler support.

Table 2-12 Methods and Classes that Support Context Handlers

Method	Description
`AccessDecision.isAccessAllowed()`	The `isAccessAllowed()` method accepts a `ContextHandler` object that can optionally be used by an access decision to obtain additional information that may be used in making the authorization decision. If the caller is unable to provide additional information, a null value should be specified.
`AdjudicatorV2.adjudicate()`	An implementation of the AdjudicatorV2 SSPI interface is the part of an adjudication provider that is called after all the Access Decisions' `isAccessAllowed` methods have been called and returned successfully (that is, without throwing exceptions). The AdjudicatorV2 SSPI accepts the resource and `ContextHandler` as additional arguments. When the AuthorizationManager calls the Adjudicator, it passes the same resource and `ContextHandler` as it passed to AccessDecision. This allows the Adjudicator to have all of the information that is available to AccessDecision.
`AuditAtnEventV2.getContext()`	Because the JAAS `LoginModule.login()` method and the `IdentityAsserter.assertIdentity()` method have access to the `ContextHandler`, the AuditAtnEventV2 interface also gets this data so it can audit relevant information. The `getContext()` method is inherited from `weblogic.security.spi.AuditContext`. The `getContext()` method gets a `ContextHandler` object from which additional audit information can be obtained.
`AuditCertPathBuilderEvent.getContext()`, `AuditCertPathValidatorEvent.getContext()`	The `getContext` method gets an optional `ContextHandler` object that may specify additional data on how to look up and validate the CertPath.
`AuditConfigurationEvent.getContext()`	The `AuditConfigurationEvent.getContext()` method gets a `ContextHandler` object from which additional audit information can be obtained.
`AuditContext.getContext()`	The `AuditContext.getContext()` method gets a `ContextHandler` object from which additional audit information can be obtained.
`AuditCredentialMappingEvent.getContext()`	The `getContext` method gets an optional `ContextHandler` object that may specify additional information about the credential mapping audit event.
`CertPathBuilderParameterSpi.getContext` and `CertPathValidatorParameterSpi.getContext`	The `CertPathBuilderParameterSpi` and `CertPathValidatorParameterSpi` interfaces include a `getContext()` method to get a `ContextHandler` that may pass in extra parameters that can be used for building and validating the Cert Path.

Table 2-12 (Cont.) Methods and Classes that Support Context Handlers

Method	Description
`ChallengeIdentityAsserterV2.assertChallengeIdentity()`, `ChallengeIdentityAsserterV2.continueChallengeIdentity()`, and `ChallengeIdentityAsserterV2.getChallengeIdentity()`	The `ChallengeIdentityAsserterV2` methods accept a `ContextHandler` object that can optionally be user by the Identity assertion provider to obtain additional information that may be used in asserting the challenge identity.
`CredentialMapperV2.getCredentials()`	The `CredentialMapper.getCredentials()` and `CredentialMapper.getCredential()` methods include a `ContextHandler` parameter with optional extra data.
`IdentityAsserterV2.assertIdentity()`	The IdentityAsserterV2 provider allows the Security Framework to pass a `ContextHandler` in the `assertIdentity` method. The `ContextHandler` object can optionally be used to obtain additional information that may be used in asserting the identity. For example, the `ContextHandler` allows users to extract extra information from the `HttpServletRequest` and to set cookies in the `HttpServletResponse`.
`LoginModule.login()`	A `ContextHandler` can be passed to the JAAS `CallbackHandler` parameter. A `CallbackHandler` is a variable-argument data structure that is passed to the `login()` method. Adding the `ContextHandler` in this manner allows users to extract extra information from the `HttpServletRequest` and to set cookies in the `HttpServletResponse`, for example. The implementation includes LoginModules used both for authentication and identity assertion. The EJB and Servlet containers must add the `ContextHandler` to the `CallbackHandler` when calling the Principal Authenticator. Specifically, they must instantiate and pass a `weblogic.security.auth.callback.ContextHandlerCallback` to the `invokeCallback` method of a `CallbackHandler` to retrieve the `ContextHandler` related to this security operation. If no `ContextHandler` is associated with this operation, `javax.security.auth.callback.UnsupportedCallbackException` is thrown.
`RoleMapper.getRoles()`	The `getRoles()` method accepts a `ContextHandler` object that can optionally be used by the role mapping provider to obtain additional information that may be used in making the authorization decision. If the caller is unable to provide additional information, a null value should be specified.

Table 2-12 (Cont.) Methods and Classes that Support Context Handlers

Method	Description
`URLCallbackHandler` and `SimpleCallbackHandler` Classes	As of WebLogic Server version 9.0, the `weblogic.security.URLCallbackHandler` and `weblogic.security.SimpleCallbackHandler` classes were updated to handle the `ContextHandler`. `URLCallbackHandler` is a `CallbackHandler` used by application developers for returning a username, password, URL, and `ContextHandler` as part of the Authenticate API. `SimpleCallbackHandler` is a simple `CallbackHandler` used by application developers for returning a username, password and `ContextHandler` as part of the Authenticate API.

Initialization of the Security Provider Database

> **Note**
>
> Prior to reviewing this section, be sure you have read Security Provider Databases in the *Understanding Security for Oracle WebLogic Server*.

At minimum, you must initialize security providers' databases with the default users, groups, security policies, security roles, or credentials that your authentication, authorization, role mapping, and credential mapping providers expect. You will need to initialize a given security provider's database *before* the security provider can be used, and should think about how this will work as you are writing the runtime classes for your custom security providers. The method you use to initialize a security provider's database depends upon many factors, including whether or not an externally administered database will be used to store the user, group, security policy, security role, or credential information, and whether or not the database already exists or needs to be created.

The following sections explain some best practices for initializing a security provider database:

- Best Practice: Create a Simple Database If None Exists
- Best Practice: Configure an Existing Database
- Best Practice: Delegate Database Initialization
- Best Practice: Use the JDBC Connection Security Service API to Obtain Database Connections

Best Practice: Create a Simple Database If None Exists

The first time an authentication, authorization, role mapping, or credential mapping provider is used, it attempts to locate a database with the information it needs to provide its security service. If the security provider fails to locate the database, you can have it create one and automatically populate it with the default users, groups, security policies, security roles, and credentials. This option may be useful for development and testing purposes.

Both the WebLogic security providers and the sample security providers follow this practice. The WebLogic Authentication, Authorization, Role Mapping, and Credential Mapping providers

store the user, group, security policy, security role, and credential information in the embedded LDAP server. If you want to use any of these WebLogic security providers, you will need to follow the Configuring the Embedded LDAP Server instructions in *Administering Security for Oracle WebLogic Server*.

> **Note**
>
> The sample security providers simply create and use a properties file as their database. For example, the sample authentication provider creates a file called `SampleAuthenticatorDatabase.java` that contains the necessary information about users and groups.

Best Practice: Configure an Existing Database

If you already have a database (such as an external LDAP server), you can populate that database with the users, groups, security policies, security roles, and credentials that your authentication, authorization, role mapping, and credential mapping providers require. (Populating an existing database is accomplished using whatever tools you already have in place for performing these tasks.)

Once your database contains the necessary information, you must configure the security providers to look in that database. You accomplish this by adding custom attributes in your security provider's MBean Definition File (MDF). Some examples of custom attributes are the database's host, port, password, and so on. After you run the MDF through the WebLogic MBeanMaker and complete a few other steps to generate the MBean type for your custom security provider, you or an administrator can use WLST or the WebLogic RESTful management resources to set these attributes to point to the database.

> **Note**
>
> See Generating an MBean Type to Configure and Manage the Custom Security Provider.

As an example, Example 2-5 shows some custom attributes that are part of the WebLogic LDAP Authentication provider's MDF. These attributes enable an administrator to specify information about the WebLogic LDAP Authentication provider's database (an external LDAP server), so it can locate information about users and groups.

Example 2-5 LDAPAuthenticator.xml

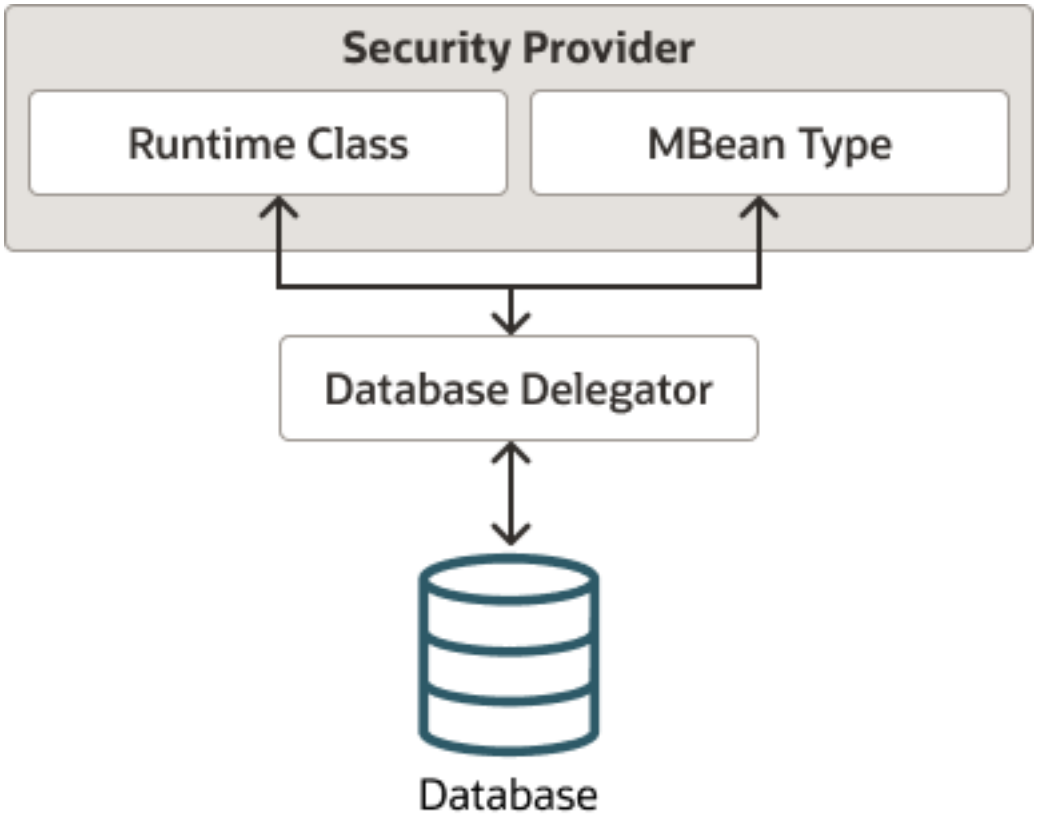
```
...
<MBeanAttribute
 Name = "UserObjectClass"
 Type = "java.lang.String"
 Default = "&quot;person&quot;"
 Description = "The LDAP object class that stores users."
/>
<MBeanAttribute
 Name = "UserNameAttribute"
 Type = "java.lang.String"
 Default = "&quot;uid&quot;"
 Description = "The attribute of an LDAP user object that specifies the name of
   the user."
```

```
/>
<MBeanAttribute
 Name = "UserDynamicGroupDNAttribute"
 Type = "java.lang.String"
 Description = "The attribute of an LDAP user object that specifies the
   distinguished names (DNs) of dynamic groups to which this user belongs.
   If such an attribute does not exist, WebLogic Server determines if a
   user is a member of a group by evaluating the URLs on the dynamic group.
   If a group contains other groups, WebLogic Server evaluates the URLs on
   any of the descendents of the group."
/>
<MBeanAttribute
 Name = "UserBaseDN"
 Type = "java.lang.String"
 Default = "&quot;ou=people, o=example.com&quot;"
 Description = "The base distinguished name (DN) of the tree in the LDAP directory
   that contains users."
/>
<MBeanAttribute
 Name = "UserSearchScope"
 Type = "java.lang.String"
 Default = "&quot;subtree&quot;"
 LegalValues = "subtree,onelevel"
 Description = "Specifies how deep in the LDAP directory tree to search for Users.
   Valid values are &lt;code&gt;subtree&lt;/code&gt;
   and &lt;code&gt;onelevel&lt;/code&gt;."
/>
...
```

Best Practice: Delegate Database Initialization

If possible, initialization calls between a security provider and the security provider's database should be done by an intermediary class, referred to as a **database delegator**. The database delegator should interact with the runtime class and the MBean type for the security provider, as shown in Figure 2-8.

Figure 2-8 Positioning of the Database Delegator Class



A database delegator is used by the WebLogic Authentication and Credential Mapping providers. The WebLogic Authentication provider, for example, calls into a database delegator to initialize the embedded LDAP server with default users and groups, which it requires to provide authentication services for the default security realm.

Use of a database delegator is suggested as a convenience to application developers and security vendors who are developing custom security providers, because it hides the security provider's database and centralizes calls into the database.

Best Practice: Use the JDBC Connection Security Service API to Obtain Database Connections

As an alternative to the best practices for creating or configuring a database for your custom security provider, you can use the `JDBCConnectionService` SSPI only during provider initialization to access the JDBC data sources that are configured for your WebLogic domain.

This capability enables your custom security providers to take advantage of full database access and database connection management capabilities provided through JDBC data sources, including multi data sources. See https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/java/sql/Connection.html for information about how SQL statements are executed and how the results are returned within the context of a connection.

When you use the `JDBCConnectionService` SSPI, note the following:

- Obtain the `JDBCConnectionService` in the `initialize()` method of your custom provider.
- Data sources are identified by name (`sqlConnectionName`), not JNDI path.

- During initialization, JDBC resources may not be available. Direct connections are returned until the JNDI and JDBC subsystems are fully initialized and available.
- When finished with the database connection returned by the JDBC data source, the security provider must invoke the `releaseConnection` method (and specify the `Connection` object) to release the connection.

Example 2-6 shows using the `JDBCConnectionService` SSPI to obtain a database connection from a named JDBC data source.

Although not shown in the example, `JDBCConnectionService.getConnection` can throw `JDBCConnectionServiceException` if the named JDBC data source is unavailable, or `SQLException` if the database connection is unavailable. `JDBCConnectionService.releaseConnection` can throw `SQLException` if the database connection is unavailable.

Example 2-6 Using the JDBCConnectionService API to Access JDBC Data Sources

```
JDBCConnectionService dbService = null;
if (services instanceof SecurityServicesJDBC) {
   try {

     dbService = ((SecurityServicesJDBC)services).getJDBCConnectionService();

     System.out.println("Obtained the JDBCConnectionService, " + dbService);

     Connection conn = dbService.getConnection("oracle-database");

     PreparedStatement statement = conn.prepareStatement("select sysdate from dual");
     ResultSet rs= statement.executeQuery();

     while (rs.next()) {
       String s1 = rs.getString(1);
       System.out.println("Sys date =" + s1);
     }

     dbService.releaseConnection(conn);
   } catch(Exception e) {
     e.printStackTrace();
   }
```

Implementing a JDBC Connection Security Service: Main Steps

To implement a security service for obtaining access to JDBC data sources:

1. In your provider's `initialize()` method, invoke the `getJDBCConnectionService` method of the `SecurityServicesJDBC` interface to obtain the JDBC connection service.
2. Invoke the `getConnection` method on the JDBC connection service instance, passing the name of a JDBC data source that is configured in your WebLogic domain.
3. Add appropriate database commands, such as prepared statements, queries, and so on.
4. You must invoke the `releaseConnection` method on the JDBC connection service instance to release the connection instance.

Differences In Attribute Validators

A validator is an interface that is implemented by a class that can validate various types of expressions. In this release of WebLogic Server, the inheritance rules for security provider attribute validator methods differ from the rules that existed in 8.1.

In 8.1, a derived MBean had only to customize an attribute validator method in its MBean implementation file to make it take effect. As of version 9.0, the derived MBean must also explicitly declare the attribute validator in its MDF file to make it take effect. Otherwise, the customized method code is ignored.

Consider the following example of the base class of all identity assert MBean implementations, `weblogic.management.security.authentication.IdentityAsserterImpl`.

IdentityAsserterImpl extends the authentication provider MBean implementation and gives the authenticator's MBean implementation access to its configuration attributes.

In 8.1, you could do the following:

1. Write an Identity Asserter provider called IdentityAsserter1. In its MDF file, indicate that it extends `weblogic.management.security.authentication.IdentityAsserter`.
2. Use the WebLogic MBeanMaker to generate the MBean type. The implementation file created by the MBeanMaker, typically named `IdentityAsserter1Impl.java`, extends `weblogic.management.security.authentication.IdentityAsserterImpl`.

   Therefore, the MBean inherits the `activeTypes` attribute, which has an attribute validator method. The `validateActiveTypes(String[] activeTypes)` method ensures that `activeTypes` includes only supported types).
3. Modify the implementation file and specify a different implementation for the `validateActiveTypes` method. For example, it could further restrict the active types or loosen the rules.
4. In 8.1, IdentityAsserter1's `validateActiveTypes` implementation is used.

   As of version 9.0, the base IdentityAsserter's `validateActiveTypes` implementation is used instead. That is, IdentityAsserter1's `validateActiveTypes` implementation is silently ignored.

To work around this difference in version 9.0 and later, redeclare the attribute validator in IdentityAsserter1's MDF file in an MBeanOperation subelement.

Differences In Attribute Validators for Custom Validators

The difference in inheritance rules for security provider attribute validators also applies to custom validators. You could have a provider declare an attribute with a custom validator. Then you could derive another provider from that one and write another implementation of the validator. In 8.1, the derived provider's validator would be used. As of version 9.0, the base provider's validator is used instead, and the derived one is silently ignored.

3

Authentication Providers

This chapter describes authentication provider concepts and functionality, and provides step-by-step instructions for developing a custom authentication provider.
Authentication is the mechanism by which callers prove that they are acting on behalf of specific users or systems. Authentication answers the question, "Who are you?" using credentials such as username/password combinations.

In WebLogic Server, authentication providers are used to prove the identity of users or system processes. Authentication providers also remember, transport, and make that identity information available to various components of a system (via subjects) when needed. During the authentication process, a principal validation provider provides additional security protections for the principals (users and groups) contained within the subject by signing and verifying the authenticity of those principals. (See Principal Validation Providers.)

This chapter includes the following sections:

- Authentication Concepts
- The Authentication Process
- Do You Need to Develop a Custom Authentication Provider?
- How to Develop a Custom Authentication Provider

> **Note**
>
> An identity assertion provider is a specific form of authentication provider that allows users or system processes to assert their identity using tokens. See Identity Assertion Providers.

Authentication Concepts

Before delving into the specifics of developing custom authentication providers, it is important to understand the following concepts:

- Users and Groups, Principals and Subjects
- LoginModules
- Java Authentication and Authorization Service (JAAS)

Users and Groups, Principals and Subjects

A **user** is similar to an operating system user in that it represents a person. A **group** is a category of users, classified by common traits such as job title. Categorizing users into groups makes it easier to control the access permissions for large numbers of users. See Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Both users and groups can be used as principals by application servers like WebLogic Server. A **principal** is an identity assigned to a user or group as a result of authentication. The Java Authentication and Authorization Service (JAAS) requires that **subjects** be used as containers for authentication information, including principals. Each principal stored in the same subject represents a separate aspect of the same user's identity, much like cards in a person's wallet. (For example, an ATM card identifies someone to their bank, while a membership card identifies them to a professional organization to which they belong.) See Java Authentication and Authorization Service (JAAS).

> **Note**
>
> Subjects replace WebLogic Server 6.x users.

Figure 3-1 illustrates the relationships among users, groups, principals, and subjects.

Figure 3-1 Relationships Among Users, Groups, Principals and Subjects

As part of a successful authentication, principals are signed and stored in a subject for future use. A principal validation provider signs principals, and an authentication provider's LoginModule actually stores the principals in the subject. Later, when a caller attempts to access a principal stored within a subject, a principal validation provider verifies that the principal has not been altered since it was signed, and the principal is returned to the caller (assuming all other security conditions are met).

> **Note**
>
> See Principal Validation Providers and LoginModules, respectively.

Any principal that is going to represent a WebLogic Server user or group needs to implement the `WLSUser` and `WLSGroup` interfaces, which are available in the `weblogic.security.spi` package.

Providing Initial Users and Groups

Authentication providers need a list of users and groups before they can be used to perform authentication in a running WebLogic Server. Some authentication providers let the administrator configure an external database (for example, add the users and groups to an LDAP server or a DBMS) and then configure the provider to use that database. These providers don't have to worry about how the users and groups are populated because the administrator does that first, using the external database's tools.

However, some authentication providers create and manage their own list of users and groups. This is the case for the ManageableSampleAuthenticator provider. These providers need to worry about how their initial set of users and groups is populated. One way to handle this is for the provider's "initialize" method to notice that the users and groups don't exist yet, and then populate the list with an initial set of users and groups.

Note that some providers have a separate list of users and groups for each security realm, and therefore need to create an initial set of users and groups the first time the list is used in a new realm. For example, the ManageableSampleAuthenticator provider creates a separate properties file of users and groups for each realm. Its `initialize` method gets the realm name, determines whether the properties file for that realm exists and, if not, creates one, populating it with its initial set of users and groups.

LoginModules

A LoginModule is a required component of an authentication provider, and can be a component of an identity assertion provider if you want to develop a separate LoginModule for perimeter authentication.

LoginModules are the work-horses of authentication: all LoginModules are responsible for authenticating users within the security realm and for populating a subject with the necessary principals (users/groups). LoginModules that are *not* used for perimeter authentication also verify the proof material submitted (for example, a user's password).

> **Note**
>
> See Identity Assertion Providers.

If there are multiple authentication providers configured in a security realm, each of the authentication providers' LoginModules will store principals within the same subject. Therefore, if a principal that represents a WebLogic Server user (that is, an implementation of the `WLSUser` interface) named "Joe" is added to the subject by one authentication provider's LoginModule, any other authentication provider in the security realm should be referring to the same person when they encounter "Joe". In other words, the other authentication providers' LoginModules should not attempt to add another principal to the subject that represents a WebLogic Server

user (for example, named "Joseph") to refer to the same person. However, it is acceptable for a another authentication provider's LoginModule to add a principal of a type other than `WLSUser` with the name "Joseph".

The LoginModule Interface

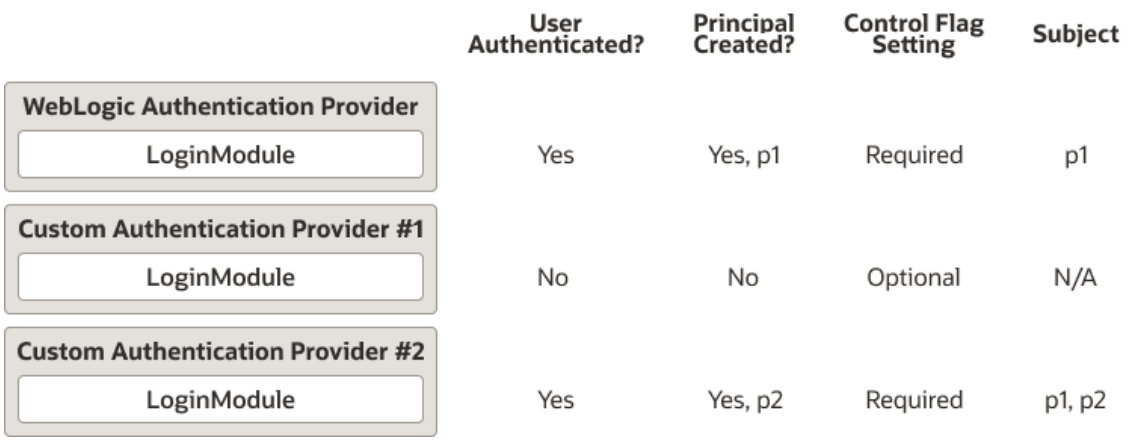
LoginModules can be written to handle a variety of authentication mechanisms, including username/password combinations, smart cards, biometric devices, and so on. You develop LoginModules by implementing the `jakarta.security.auth.spi.LoginModule` interface, which is based on the Java Authentication and Authorization Service (JAAS) and uses a subject as a container for authentication information. The `LoginModule` interface enables you to plug in different kinds of authentication technologies for use with a single application, and the WebLogic Security Framework is designed to support multiple `LoginModule` implementations for multipart authentication. You can also have dependencies across LoginModule instances or share credentials across those instances. However, the relationship between LoginModules and authentication providers is one-to-one. In other words, to have a LoginModule that handles retina scan authentication and a LoginModule that interfaces to a hardware device like a smart card, you must develop and configure two authentication providers, each of which include an implementation of the `LoginModule` interface. See Implement the JAAS LoginModule Interface.

> **Note**
>
> You can also obtain LoginModules from third-party security vendors instead of developing your own.

LoginModules and Multipart Authentication

The way you configure multiple authentication providers (and thus, multiple LoginModules) can affect the overall outcome of the authentication process, which is especially important for multipart authentication. First, because LoginModules are components of authentication providers, they are called in the order in which the authentication providers are configured. Generally, you configure authentication providers using the WebLogic Remote Console. (See Specifying the Order of Authentication Providers.) Second, the way each LoginModule's control flag is set specifies how a failure during the authentication process should be handled. Figure 3-2 illustrates a sample flow involving three different LoginModules (that are part of three authentication providers), and illustrates what happens to the subject for different authentication outcomes.

Figure 3-2 Sample LoginModule Flow



If the control flag for Custom Authentication Provider #1 had been set to Required, the authentication failure in its User Authentication step would have caused the entire authentication process to have failed. Also, if the user had not been authenticated by the WebLogic Authentication provider (or custom authentication provider #2), the entire authentication process would have failed. If the authentication process had failed in any of these ways, all three LoginModules would have been rolled back and the subject would not contain any principals.

> **Note**
>
> See the Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide in *Security Developer's Guide* and the LoginModule interface (`http://docs.oracle.com/javase/8/docs/api/javax/security/auth/spi/LoginModule.html`), respectively.

Java Authentication and Authorization Service (JAAS)

Whether the client is an application, applet, Enterprise JavaBean (EJB), or servlet that requires authentication, WebLogic Server uses the Java Authentication and Authorization Service (JAAS) classes to reliably and securely authenticate to the client. JAAS implements a Java version of the Pluggable Authentication Module (PAM) framework, which permits applications to remain independent from underlying authentication technologies. Therefore, the PAM framework allows the use of new or updated authentication technologies without requiring modifications to your application.

WebLogic Server uses JAAS for remote fat-client authentication, and internally for authentication. Therefore, only developers of custom authentication providers and developers of remote fat client applications need to be involved with JAAS directly. Users of thin clients or developers of within-container fat client applications (for example, those calling an Enterprise JavaBean (EJB) from a servlet) do not require the direct use or knowledge of JAAS.

How JAAS Works With the WebLogic Security Framework

Generically, authentication using the JAAS classes and WebLogic Security Framework is performed in the following manner:

1. A client-side application obtains authentication information from a user or system process. The mechanism by which this occurs is different for each type of client.
2. The client-side application can optionally create a `CallbackHandler` containing the authentication information.
   a. The client-side application passes the `CallbackHandler` to a local (client-side) LoginModule using the `LoginContext` class. (The local LoginModule could be `UsernamePasswordLoginModule`, which is provided as part of WebLogic Server.)
   b. The local LoginModule passes the `CallbackHandler` containing the authentication information to the appropriate WebLogic Server container (for example, RMI, EJB, servlet, or IIOP).

> **Note**
>
> A CallbackHandler is a highly-flexible JAAS standard that allows a variable number of arguments to be passed as complex objects to a method. There are three types of CallbackHandlers: NameCallback, PasswordCallback, and TextInputCallback, all of which reside in the `jakarta.security.auth.callback` package. The NameCallback and PasswordCallback return the username and password, respectively. TextInputCallback can be used to access the data users enter into any additional fields on a login form (that is, fields other than those for obtaining the username and password). When used, there should be one TextInputCallback per additional form field, and the prompt string of each TextInputCallback must match the field name in the form. WebLogic Server only uses the TextInputCallback for form-based Web application login. See the CallbackHandler interface (https://jakarta.ee/specifications/authentication/2.0/jakarta-authentication-spec-2.0#a608).
>
> See the LoginModulet class (https://jakarta.ee/specifications/authentication/2.0/jakarta-authentication-spec-2.0#loginmodule-bridge-profile).
>
> See *Java API Reference for Oracle WebLogic Server* for the UsernamePasswordLoginModule class.
>
> If you do not want to use a client-side LoginModule, you can specify the username and password in other ways: for example, as part of the initial JNDI lookup.

3. The WebLogic Server container calls into the WebLogic Security Framework. If there is a client-side `CallbackHandler` containing authentication information, this is passed into the WebLogic Security Framework.
4. For each of the configured authentication providers, the WebLogic Security Framework creates a `CallbackHandler` using the authentication information that was passed in. (These are internal `CallbackHandlers` created on the server-side by the WebLogic Security Framework, and are not related to the client's `CallbackHandler`.)
5. The WebLogic Security Framework calls the LoginModule associated with the authentication provider (that is, the LoginModule that is specifically designed to handle the authentication information).

   > **Note**
   >
   > See LoginModules.

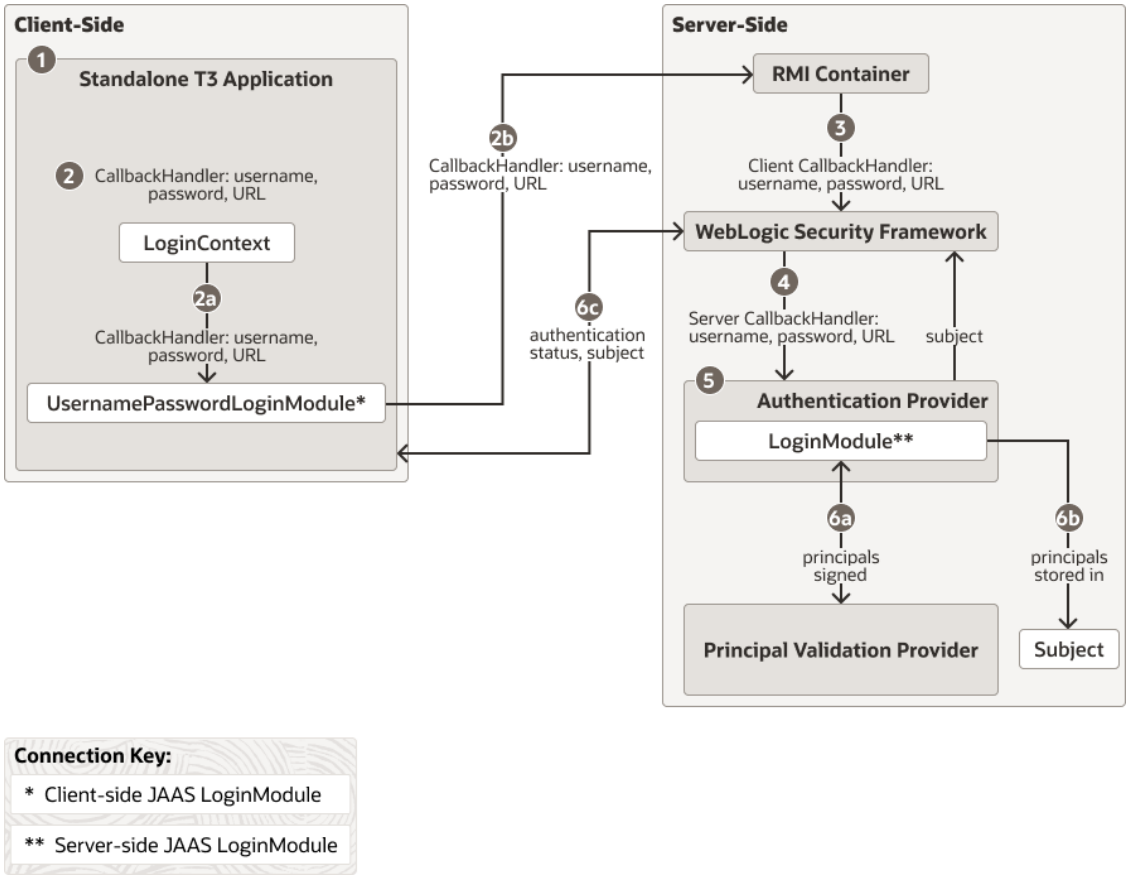
   The LoginModule attempts to authenticate the client using the authentication information.
6. If the authentication is successful, the following occurs:
   a. Principals (users and groups) are signed by a principal validation provider to ensure their authenticity between programmatic server invocations. See Principal Validation Providers.
   b. The LoginModule associates the signed principals with a subject, which represents the user or system process being authenticated. See Users and Groups, Principals and Subjects.

> **Note**
>
> For authentication performed entirely on the server-side, the process would begin at step 3, and the WebLogic Server container would call the `weblogic.security.services.authentication.login` method prior to step 4.

Example: Standalone T3 Application

Figure 3-3 illustrates how the JAAS classes work with the WebLogic Security Framework for a standalone, T3 application, and an explanation follows.

Figure 3-3 Authentication Using JAAS Classes and WebLogic Server



For this example, authentication using the JAAS classes and WebLogic Security Framework is performed in the following manner:

1. The T3 application obtains authentication information (username, password, and URL) from a user or system process.
2. The T3 application creates a `CallbackHandler` containing the authentication information.
   a. The T3 application passes the `CallbackHandler` to the `UsernamePasswordLoginModule` using the `LoginContext` class.

> **Note**
>
> The `weblogic.security.auth.login.UsernamePasswordLoginModule` implements the standard JAAS `jakarta.security.auth.spi.LoginModule` interface and uses client-side APIs to authenticate a WebLogic client to a WebLogic Server instance. It can be used for both T3 and IIOP clients. Callers of this LoginModule must implement a CallbackHandler to pass the username (NameCallback), password (PasswordCallback), and a URL (URLCallback).

   b. The `UsernamePasswordLoginModule` passes the `CallbackHandler` containing the authentication information (that is, username, password, and URL) to the WebLogic Server RMI container.

3. The WebLogic Server RMI container calls into the WebLogic Security Framework. The client-side `CallbackHandler` containing authentication information is passed into the WebLogic Security Framework.
4. For each of the configured authentication providers, the WebLogic Security Framework creates a `CallbackHandler` containing the username, password, and URL that was passed in. (These are internal `CallbackHandlers` created on the server-side by the WebLogic Security Framework, and are not related to the client's `CallbackHandler`.)
5. The WebLogic Security Framework calls the LoginModule associated with the authentication provider (that is, the LoginModule that is specifically designed to handle the authentication information).

   The LoginModule attempts to authenticate the client using the authentication information.
6. If the authentication is successful, the following occurs:

   a. Principals (users and groups) are signed by a principal validation provider to ensure their authenticity between programmatic server invocations.

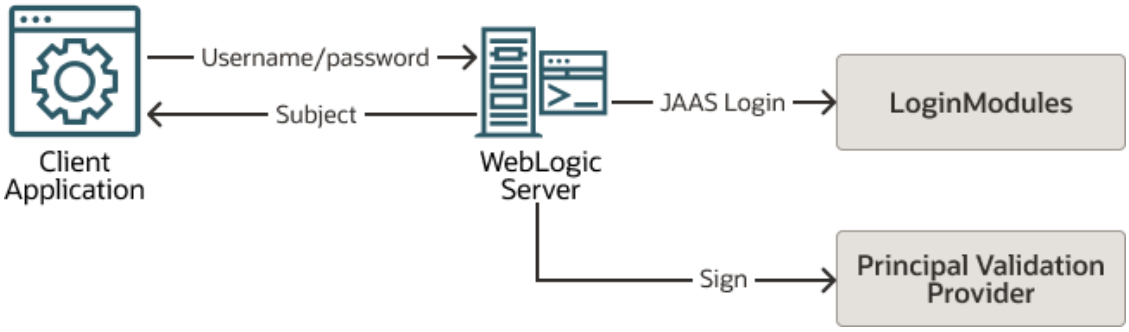
   b. The LoginModule associates the signed principals with a subject, which represents the user or system being authenticated.

   c. The WebLogic Security Framework returns the authentication status to the T3 client application, and the T3 client application retrieves the authenticated subject from the WebLogic Security Framework.

The Authentication Process

Figure 3-4 shows a behind-the-scenes look of the authentication process for a fat-client login. JAAS runs on the server to perform the login. Even in the case of a thin-client login (that is, a browser client) JAAS is still run on the server.

Figure 3-4 The Authentication Process



> **Note**
>
> Only developers of custom authentication providers will be involved with this JAAS process directly. The client application could either use JNDI initial context creation or JAAS to initiate the passing of the username and password.

When a user attempts to log into a system using a username/password combination, WebLogic Server establishes trust by validating that user's username and password, and returns a subject that is populated with principals per JAAS requirements. As Figure 3-4 also shows, this process requires the use of a LoginModule and a principal validation provider, which are discussed in detail in LoginModules and Principal Validation Providers respectively.

After successfully proving a caller's identity, an authentication context is established, which allows an identified user or system to be authenticated to other entities. Authentication contexts may also be delegated to an application component, allowing that component to call another application component while impersonating the original caller.

Do You Need to Develop a Custom Authentication Provider?

The default (that is, active) security realm for WebLogic Server includes a WebLogic Authentication provider.

> **Note**
>
> In conjunction with the WebLogic Authorization provider, the WebLogic Authentication provider replaces the functionality of the File realm that was available in 6.x releases of WebLogic Server.

The WebLogic Authentication provider supports delegated username/password authentication, and utilizes an embedded LDAP server to store user and group information. The WebLogic Authentication provider allows you to edit, list, and manage users and group membership.

WebLogic Server also provides the following additional authentication providers that you can use instead of or in conjunction with the WebLogic Authentication provider in the default security realm:

- A set of LDAP authentication providers that access external LDAP stores (including Open LDAP and Microsoft Active Directory).
- A set of Database Base Management System (DBMS) authentication providers that access user, password, group, and group membership information stored in databases for authentication
- An LDAP X509 Identity Assertion provider.

By default, these additional authentication providers are available but not configured in the WebLogic default security realm.

If you want to perform additional authentication tasks, then you need to develop a custom authentication provider.

> **Note**
>
> If you want to perform perimeter authentication using a token type that is not supported out of the box (for example, a new, custom, or third party token type), you might need to develop a custom identity assertion provider. See Identity Assertion Providers.

How to Develop a Custom Authentication Provider

If the WebLogic Authentication provider does not meet your needs, you can develop a custom authentication provider by following these steps:

1. Create Runtime Classes Using the Appropriate SSPIs
2. Generate an MBean type for your custom authentication provider by completing the steps described in Generate an MBean Type Using the WebLogic MBeanMaker.
3. Configure the Custom Authentication Provider

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom authentication provider by following these steps:

- Implement the AuthenticationProviderV2 SSPI
- Implement the JAAS LoginModule Interface

For an example of how to create a runtime class for a custom authentication provider, see Example: Creating the Runtime Classes for the Sample Authentication Provider .

Implement the AuthenticationProviderV2 SSPI

> **Note**
>
> The `AuthenticationProvider` SSPI is deprecated in this release of WebLogic Server. Use the `AuthenticationProviderV2` SSPI instead.

To implement the `AuthenticationProviderV2` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following methods:

- `getLoginModuleConfiguration`

  ```
  public AppConfigurationEntry getLoginModuleConfiguration()
  ```

  The `getLoginModuleConfiguration` method obtains information about the authentication provider's associated LoginModule, which is returned as an `AppConfigurationEntry`. The `AppConfigurationEntry` is a Java Authentication and Authorization Service (JAAS) class that contains the classname of the LoginModule; the LoginModule's control flag (which was passed in via the authentication provider's associated MBean); and a configuration options map for the LoginModule (which allows other configuration information to be passed into the LoginModule).

  To know about the `AppConfigurationEntry` class (located in the `jakarta.security.auth.login` package) and the control flag options for LoginModules, see the AppConfigurationEntry class (`http://docs.oracle.com/javase/8/docs/api/javax/security/auth/login/AppConfigurationEntry.html`) and the Configuration class (`http://docs.oracle.com/javase/8/docs/api/javax/security/auth/login/Configuration.html`). See LoginModulesand Understand Why You Need an MBean Type .

- `getAssertionModuleConfiguration`

  ```
  public AppConfigurationEntry
  getAssertionModuleConfiguration()
  ```

  The `getAssertionModuleConfiguration` method obtains information about an identity assertion provider's associated LoginModule, which is returned as an `AppConfigurationEntry`. The `AppConfigurationEntry` is a JAAS class that contains the classname of the LoginModule; the LoginModule's control flag (which was passed in via the identity assertion provider's associated MBean); and a configuration options map for the LoginModule (which allows other configuration information to be passed into the LoginModule).

> **Note**
>
> The implementation of the `getAssertionModuleConfiguration` method can be to return null, if you want the identity assertion provider to use the same LoginModule as the authentication provider.
>
> The `assertIdentity()` method of an identity assertion provider is called every time identity assertion occurs, but the LoginModules may not be called if the Subject is cached. The `-Dweblogic.security.identityAssertionTTL` flag can be used to affect this behavior (for example, to modify the default TTL of 5 minutes or to disable the cache by setting the flag to -1).
>
> It is the responsibility of the identity assertion provider to ensure not just that the token is valid, but also that the user is still valid (for example, the user has not been deleted).
>
> To use the EJB <run-as-principal> element with a custom authentication provider, use the `getAssertionModuleConfiguration()` method. This method performs the identity assertion that validates the principal specified in the <run-as-principal>element.

- `getPrincipalValidator`

  ```
  public PrincipalValidator getPrincipalValidator()
  ```

  The `getPrincipalValidator` method obtains a reference to the principal validation provider's runtime class (that is, the `PrincipalValidator` SSPI implementation). In most cases, the WebLogic Principal Validation provider can be used (see Example 3-1 for an example of how to return the WebLogic Principal Validation provider). See Principal Validation Providers.

- `getIdentityAsserter`

  ```
  public IdentityAsserterV2 getIdentityAsserter()
  ```

  The AuthenticationProviderV2 `getIdentityAsserter` method obtains a reference to the new identity assertion provider's runtime class (that is, the `IdentityAsserterV2` SSPI implementation).

  In most cases, the return value for this method will be `null` (see Example 3-1 for an example). See Identity Assertion Providers.

See *Java API Reference for Oracle WebLogic Server* to know more about the `AuthenticationProviderV2` SSPI and the methods described above.

Implement the JAAS LoginModule Interface

To implement the JAAS `jakarta.security.auth.spi.LoginModule` interface, provide implementations for the following methods:

- `initialize`

  ```
  public void initialize (Subject subject, CallbackHandler callbackHandler, Map
  sharedState, Map options)
  ```

  The `initialize` method initializes the LoginModule. It takes as arguments a subject in which to store the resulting principals, a `CallbackHandler` that the authentication provider will use to call back to the container for authentication information, a map of any shared

state information, and a map of configuration options (that is, any additional information you want to pass to the LoginModule).

A `CallbackHandler` is a highly-flexible JAAS standard that allows a variable number of arguments to be passed as complex objects to a method. See the *Jakarta 9.1 API Specification* for the `CallbackHandler` interface (https://jakarta.ee/specifications/platform/9.1/apidocs/jakarta/security/auth/message/callback/package-frame).

- `login`

  ```
  public boolean login() throws LoginException
  ```

  The `login` method attempts to authenticate the user and create principals for the user by calling back to the container for authentication information. If multiple LoginModules are configured (as part of multiple authentication providers), this method is called for each LoginModule in the order that they are configured. Information about whether the login was successful (that is, whether principals were created) is stored for each LoginModule.

- `commit`

  ```
  public boolean commit() throws LoginException
  ```

  The `commit` method attempts to add the principals created in the `login` method to the subject. This method is also called for each configured LoginModule (as part of the configured authentication providers), and executed in order. Information about whether the commit was successful is stored for each LoginModule.

- `abort`

  ```
  public boolean abort() throws LoginException
  ```

  The `abort` method is called for each configured LoginModule (as part of the configured authentication providers) if any commits for the LoginModules failed (in other words, the relevant `REQUIRED`, `REQUISITE`, `SUFFICIENT` and `OPTIONAL` LoginModules did not succeed). The `abort` method will remove that LoginModule's principals from the subject, effectively rolling back the actions performed. See the LoginModule interface (https://jakarta.ee/specifications/platform/9.1/apidocs/jakarta/security/enterprise/package-frame).

- `logout`

  ```
  public boolean logout() throws LoginException
  ```

  The `logout` method attempts to log the user out of the system. It also resets the subject so that its associated principals are no longer stored.

> **Note**
>
> The `LoginModule.logout` method is never called for the WebLogic Authentication providers or custom authentication providers. This is simply because once the principals are created and placed into a subject, the WebLogic Security Framework no longer controls the lifecycle of the subject. Therefore, the developer-written, user code that creates the JAAS LoginContext to login and obtain the subject should also call the `LoginContext.logout` method. When the user code runs in a Java client that uses JAAS directly, that code has the option of calling the `LoginContext.logout` method, which clears the subject. When the user code runs in a servlet, the servlet has the ability to logout a user from a servlet session, which clears the subject.

See the Jakarta Authentication and Authorization Service Developer's Guide (https://jakarta.ee/learn/specification-guides/security-authorization-and-authentication-explained/) and the LoginModule interface (https://jakarta.ee/specifications/platform/9.1/apidocs/jakarta/security/enterprise/package-frame).

Throwing Custom Exceptions from LoginModules

You may want to throw a custom exception from a LoginModule you write. The custom exception can then be caught by your application and appropriate action taken. For example, if a `PasswordChangeRequiredException` is thrown from your LoginModule, you can catch that exception within your application, and use it to forward users to a page that allows them to change their password.

When you throw a custom exception from a LoginModule and want to catch it within your application, you must ensure that:

1. The application catching the exception is running on the server. (Fat clients cannot catch custom exceptions.)
2. Your servlet has access to the custom exception class at both compile time and deploy time. You can do this using either of the following methods, depending on your preference:
   - Method 1: Make Custom Exceptions Available via the System and Compiler Classpath
   - Method 2: Make Custom Exceptions Available via the Application Classpath

Method 1: Make Custom Exceptions Available via the System and Compiler Classpath

1. Write an exception class that extends `LoginException`.
2. Use the custom exception class in your classes that implement the `LoginModule` and `AuthenticationProvider` interfaces.
3. Put the custom exception class in both the system and compiler classpath when compiling the security provider's runtime class.
4. Generate an MBean type for your custom authentication provider, as explained in Generate an MBean Type Using the WebLogic MBeanMaker.

Method 2: Make Custom Exceptions Available via the Application Classpath

1. Write an exception class that extends `LoginException`.
2. Use the custom exception class in your classes that implement the `LoginModule` and `AuthenticationProvider` interfaces.
3. Put the custom exception's source in the classpath of the application's build, and include it in the classpath of the application's JAR/WAR file.
4. Generate an MBean type for your custom authentication provider, as explained in Generate an MBean Type Using the WebLogic MBeanMaker.
5. Add the custom exception class to the MJF (MBean JAR File) generated by the WebLogic MBeanMaker.
6. Include the MJF when compiling your application.

Example: Creating the Runtime Classes for the Sample Authentication Provider

Example 3-1 shows the `SimpleSampleAuthenticationProviderImpl.java` class, which is one of two runtime classes for the sample authentication provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription` and `shutdown` (as described in Understand the Purpose of the Provider SSPIs.)
- The four methods in the `AuthenticationProviderV2` SSPI: the `getLoginModuleConfiguration`, `getAssertionModuleConfiguration`, `getPrincipalValidator`, and `getIdentityAsserter` methods (as described in Implement the AuthenticationProviderV2 SSPI).

> **Note**
>
> The boldface code in Example 3-1 highlights the class declaration and the method signatures.

Example 3-1 SimpleSampleAuthenticationProviderImpl.java

```
package examples.security.providers.authentication.simple;
import java.util.HashMap;
import jakarta.security.auth.login.AppConfigurationEntry;
import jakarta.security.auth.login.AppConfigurationEntry.LoginModuleControlFlag;
import weblogic.management.security.ProviderMBean;
import weblogic.security.spi.AuthenticationProviderV2;
import weblogic.security.spi.IdentityAsserterV2;
import weblogic.security.spi.PrincipalValidator;
import weblogic.security.spi.SecurityServices;
import weblogic.security.principal.WLSGroupImpl;
import weblogic.security.principal.WLSUserImpl;
public final class SimpleSampleAuthenticationProviderImpl implements AuthenticationProviderV2
{
   private String description;
   private SimpleSampleAuthenticatorDatabase database;
   private LoginModuleControlFlag controlFlag;
   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SimpleSampleAuthenticationProviderImpl.initialize");
      SimpleSampleAuthenticatorMBean myMBean = (SimpleSampleAuthenticatorMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
      database = new SimpleSampleAuthenticatorDatabase(myMBean);
      String flag = myMBean.getControlFlag();
      if (flag.equalsIgnoreCase("REQUIRED")) {
        controlFlag = LoginModuleControlFlag.REQUIRED;
      } else if (flag.equalsIgnoreCase("OPTIONAL")) {
        controlFlag = LoginModuleControlFlag.OPTIONAL;
      } else if (flag.equalsIgnoreCase("REQUISITE")) {
        controlFlag = LoginModuleControlFlag.REQUISITE;
      } else if (flag.equalsIgnoreCase("SUFFICIENT")) {
        controlFlag = LoginModuleControlFlag.SUFFICIENT;
      } else {
        throw new IllegalArgumentException("invalid flag value" + flag);
      }
   }
   public String getDescription()
```

```
   {
      return description;
   }
   public void shutdown()
   {
      System.out.println("SimpleSampleAuthenticationProviderImpl.shutdown");
   }
   private AppConfigurationEntry getConfiguration(HashMap options)
   {
      options.put("database", database);
      return new
        AppConfigurationEntry(
          "examples.security.providers.authentication.Simple.Simple.SampleLoginModuleImpl",
          controlFlag,
          options
        );
   }
   public AppConfigurationEntry getLoginModuleConfiguration()
   {
      HashMap options = new HashMap();
      return getConfiguration(options);
   }
   public AppConfigurationEntry getAssertionModuleConfiguration()
   {
      HashMap options = new HashMap();
      options.put("IdentityAssertion","true");
      return getConfiguration(options);
   }
   public PrincipalValidator getPrincipalValidator()
   {
      return new PrincipalValidatorImpl();
   }
   public IdentityAsserterV2 getIdentityAsserter()
   {
      return null;
   }
}
```

Example 3-2 shows the `SampleLoginModuleImpl.java` class, which is one of two runtime classes for the sample authentication provider. This runtime class implements the JAAS LoginModule interface (as described in Implement the JAAS LoginModule Interface), and therefore includes implementations for its `initialize`, `login`, `commit`, `abort`, and `logout` methods.

> **Note**
>
> The boldface code in Example 3-2 highlights the class declaration and the method signatures.

Example 3-2 SimpleSampleLoginModuleImpl.java

```
package examples.security.providers.authentication.simple;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Map;
import java.util.Vector;
import jakarta.security.auth.Subject;
import jakarta.security.auth.callback.Callback;
import jakarta.security.auth.callback.CallbackHandler;
```

```
import jakarta.security.auth.callback.NameCallback;
import jakarta.security.auth.callback.PasswordCallback;
import jakarta.security.auth.callback.UnsupportedCallbackException;
import jakarta.security.auth.login.LoginException;
import jakarta.security.auth.login.FailedLoginException;
import jakarta.security.auth.spi.LoginModule;
import weblogic.management.utils.NotFoundException;
import weblogic.security.spi.WLSGroup;
import weblogic.security.spi.WLSUser;
import weblogic.security.principal.WLSGroupImpl;
import weblogic.security.principal.WLSUserImpl;
final public class SimpleSampleLoginModuleImpl implements LoginModule
{
   private Subject subject;
   private CallbackHandler callbackHandler;
   private SimpleSampleAuthenticatorDatabase database;
   // Determine whether this is a login or assert identity
   private boolean isIdentityAssertion;
   // Authentication status
   private boolean loginSucceeded;
   private boolean principalsInSubject;
   private Vector principalsForSubject = new Vector();
   public void initialize(Subject subject, CallbackHandler callbackHandler, Map
   sharedState, Map options)
   {
      // only called (once!) after the constructor and before login
      System.out.println("SimpleSampleLoginModuleImpl.initialize");
      this.subject = subject;
      this.callbackHandler = callbackHandler;
      // Check for Identity Assertion option
      isIdentityAssertion =
         "true".equalsIgnoreCase((String)options.get("IdentityAssertion"));
      database = (SimpleSampleAuthenticatorDatabase)options.get("database");
   }
   public boolean login() throws LoginException
   {
      // only called (once!) after initialize
      System.out.println("SimpleSampleLoginModuleImpl.login");
      // loginSucceeded       should be false
      // principalsInSubject  should be false

      Callback[] callbacks = getCallbacks();
      String userName = getUserName(callbacks);
      if (userName.length() > 0) {
         if (!database.userExists(userName)) {
            throwFailedLoginException("Authentication Failed: User " + userName
            + " doesn't exist.");
         }
        if (!isIdentityAssertion) {
         String passwordWant = null;
         try {
            passwordWant = database.getUserPassword(userName);
         } catch (NotFoundException shouldNotHappen) {}
            String passwordHave = getPasswordHave(userName, callbacks);
            if (passwordWant == null || !passwordWant.equals(passwordHave)) {
               throwFailedLoginException(
                 "Authentication Failed: User " + userName + " bad password."
               );
            }
         }
         } else {
          // anonymous login - let it through?
```

```
        System.out.println("\tempty userName");
        }
        loginSucceeded = true;
        principalsForSubject.add(new WLSUserImpl(userName));
        addGroupsForSubject(userName);
        return loginSucceeded;
    }
    public boolean commit() throws LoginException
    {
       // only called (once!) after login
       // loginSucceeded      should be true or false
       // principalsInSubject should be false
       // user      should be null if !loginSucceeded, null or not-null otherwise
       // group     should be null if user == null, null or not-null otherwise

       System.out.println("SimpleSampleLoginModule.commit");
       if (loginSucceeded) {
          subject.getPrincipals().addAll(principalsForSubject);
          principalsInSubject = true;
          return true;
       } else {
          return false;
       }
    }
    public boolean abort() throws LoginException
    {
       // The abort method is called to abort the authentication process. This is
       // phase 2 of authentication when phase 1 fails. It is called if the
       // LoginContext's overall authentication failed.
       // loginSucceeded      should be true or false
       // user      should be null if !loginSucceeded, otherwise null or not-null
       // group     should be null if user == null, otherwise null or not-null
       // principalsInSubject      should be false if user is null, otherwise true
       //                          or false

       System.out.println("SimpleSampleLoginModule.abort");
       if (principalsInSubject) {
          subject.getPrincipals().removeAll(principalsForSubject);
          principalsInSubject = false;
       }
       return true;
    }
    public boolean logout() throws LoginException
    {
       // should never be called
       System.out.println("SimpleSampleLoginModule.logout");
       return true;
    }
    private void throwLoginException(String msg) throws LoginException
    {
       System.out.println("Throwing LoginException(" + msg + ")");
       throw new LoginException(msg);
    }
    private void throwFailedLoginException(String msg) throws FailedLoginException
    {
       System.out.println("Throwing FailedLoginException(" + msg + ")");
       throw new FailedLoginException(msg);
    }
    private Callback[] getCallbacks() throws LoginException
    {
       if (callbackHandler == null) {
          throwLoginException("No CallbackHandler Specified");
```

```
      }
      if (database == null) {
         throwLoginException("database not specified");
      }
      Callback[] callbacks;
      if (isIdentityAssertion) {
         callbacks = new Callback[1];
      } else {
         callbacks = new Callback[2];
         callbacks[1] = new PasswordCallback("password: ",false);
      }
      callbacks[0] = new NameCallback("username: ");
      try {
          callbackHandler.handle(callbacks);
      } catch (IOException e) {
         throw new LoginException(e.toString());
      } catch (UnsupportedCallbackException e) {
         throwLoginException(e.toString() + " " + e.getCallback().toString());
      }
      return callbacks;
   }
   private String getUserName(Callback[] callbacks) throws LoginException
   {
      String userName = ((NameCallback)callbacks[0]).getName();
      if (userName == null) {
         throwLoginException("Username not supplied.");
      }
      System.out.println("\tuserName\t= " + userName);
      return userName;
   }
   private void addGroupsForSubject(String userName)
   {
      for (Enumeration e = database.getUserGroups(userName);
         e.hasMoreElements();) {
            String groupName = (String)e.nextElement();
            System.out.println("\tgroupName\t= " + groupName);
            principalsForSubject.add(new WLSGroupImpl(groupName));
      }
   }
   private String getPasswordHave(String userName, Callback[] callbacks) throws
   LoginException
   {
      PasswordCallback passwordCallback = (PasswordCallback)callbacks[1];
      char[] password = passwordCallback.getPassword();
      passwordCallback.clearPassword();
      if (password == null || password.length < 1) {
         throwLoginException("Authentication Failed: User " + userName + ".
            Password not supplied");
      }
      String passwd = new String(password);
      System.out.println("\tpasswordHave\t= " + passwd);
      return passwd;
   }
}
```

Configure the Custom Authentication Provider

Configuring a custom authentication provider means that you are adding the custom authentication provider to your security realm, where it can be accessed by applications requiring authentication services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers. This section contains information that is important for the person configuring your custom authentication providers:

- Managing User Lockouts
- Specifying the Order of Authentication Providers

> **Note**
>
> The steps for configuring a custom authentication provider are described in Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

Managing User Lockouts

As part of using a custom authentication provider, you need to consider how you will configure and manage user lockouts. You have two choices for doing this:

- Rely on the Realm-Wide User Lockout Manager
- Implement Your Own User Lockout Manager

Rely on the Realm-Wide User Lockout Manager

The WebLogic Security Framework provides a realm-wide User Lockout Manager that works directly with the WebLogic Security Framework to manage user lockouts.

> **Note**
>
> Both the realm-wide User Lockout Manager and a WebLogic Server `PasswordValidatorMBean` (at the realm level) may be active. See *Java API Reference for Oracle WebLogic Server*.

If you decide to rely on the realm-wide User Lockout Manager, then all you must do to make it work with your custom authentication provider is use the WebLogic Remote Console to:

1. Ensure that User Lockout is enabled. (It should be enabled by default.)
2. Modify any parameters for User Lockout (as necessary).

> **Note**
>
> Changes to the User Lockout Manager do not take effect until you reboot the server. Instructions for using the WebLogic Remote Console to perform these tasks are described in Protecting User Accounts in *Administering Security for Oracle WebLogic Server*.

Implement Your Own User Lockout Manager

If you decide to implement your own User Lockout Manager as part of your custom authentication provider, then you must:

1. Disable the realm-wide User Lockout Manager to prevent double lockouts from occurring. (When you create a new security realm using the WebLogic Remote Console, a User Lockout Manager is always created.) Instructions for performing this task are provided in Protecting User Accounts in *Administering Security for Oracle WebLogic Server*.
2. Because you cannot borrow anything from the WebLogic Security Framework's realm-wide implementation, you must also perform the following tasks:
   a. Provide the implementation for your User Lockout Manager. Note that there is no security service provider interface (SSPI) provided for User Lockout Managers.
   b. Modify an MBean by which the User Lockout Manager can be managed.

Specifying the Order of Authentication Providers

As described in LoginModules and Multipart Authentication, the order in which you configure multiple authentication providers (and thus LoginModules) affects the outcome of the authentication process.

You can configure authentication providers in any order. However, if you need to reorder your configured authentication providers, follow the steps described in Changing the Order of Authentication Providers in *Administering Security for Oracle WebLogic Server*.

4

Identity Assertion Providers

This chapter describes identity assertion provider concepts and functionality, and provides step-by-step instructions for developing a custom identity assertion provider.
An identity assertion provider is a specific form of authentication provider that allows users or system processes to assert their identity using tokens (in other words, perimeter authentication). identity assertion providers enable perimeter authentication and support single sign-on. You can use an identity assertion provider in place of an authentication provider if you create a LoginModule for the identity assertion provider, or in addition to an authentication provider if you want to use the authentication provider's LoginModule.

If you want to allow the identity assertion provider to be configured separately from the authentication provider, write two providers. If your identity assertion provider and authentication provider cannot work independently, then write one provider.

This chapter includes the following sections:

Identity Assertion Concepts

Before you develop an identity assertion provider, you need to understand the following concepts:

Identity Assertion Providers and LoginModules

When used with a LoginModule, identity assertion providers support single sign-on. For example, an identity assertion provider can generate a token from a digital certificate, and that token can be passed around the system so that users are not asked to sign on more than once.

The LoginModule that an identity assertion provider uses can be:

- Part of a custom authentication provider you develop. For more information, see Authentication Providers.
- Part of the WebLogic Authentication provider Oracle developed and packaged with WebLogic Server. See Do You Need to Develop a Custom Authentication Provider?.
- Part of a third-party security vendor's authentication provider.

Unlike in a simple authentication situation (described in The Authentication Process), the LoginModules that identity assertion providers use *do not* verify proof material such as usernames and passwords; they simply verify that the user exists.

The LoginModules in this configuration must:

- Populate the Subject with required Principals, such as those of type WLSGroup.
- Must trust that the user has submitted sufficient proof to login and not require a password or some other proof material.

You must implement the `AuthenticationProviderV2.getAssertionModuleConfiguration` method in your custom authentication provider, as described in Implement the AuthenticationProviderV2 SSPI. This method is called for identity assertion, such as when an X.509 certificate is being used, and to process the run-as tag in deployment descriptors. Other single signon strategies use it as well.

> **Note**
>
> See LoginModules.

Identity Assertion and Tokens

You develop identity assertion providers to support the specific types of tokens that you will be using to assert the identities of users or system processes. You can develop an identity assertion provider to support multiple token types, but you or an administrator configure the identity assertion provider so that it validates only one `active` token type. While you can have multiple identity assertion providers in a security realm with *the ability* to validate the same token type, only one identity assertion provider can actually perform this validation.

> **Note**
>
> *Supporting* token types means that the identity assertion provider's runtime class (that is, the `IdentityAsserter` SSPI implementation) can validate the token type in its `assertIdentity` method. See Implement the IdentityAsserterV2 SSPI.

The following sections will help you work with new token types:

- How to Create New Token Types
- How to Make New Token Types Available for Identity Assertion Provider Configurations

How to Create New Token Types

If you develop a custom identity assertion provider, you can also create new token types. A **token type** is simply a piece of data represented as a string. The token types you create and use are completely up to you. The token types currently defined for the WebLogic Identity Assertion provider include: `AuthenticatedUser`, `X.509`, `CSI.PrincipalName`, `CSI.ITTAnonymous`, `CSI.X509CertChain`, `CSI.DistinguishedName`, and `wsse:PasswordDigest`.

To create new token types, you create a new Java file and declare any new token types as variables of type `String`., as shown in Example 4-1. The

`PerimeterIdentityAsserterTokenTypes.java` file defines the names of the token types `Test 1`, `Test 2`, and `Test 3` as strings.

> **Note**
>
> If you are defining only one new token type, you can also do it right in the identity assertion provider's runtime class, as shown in Example 4-4.

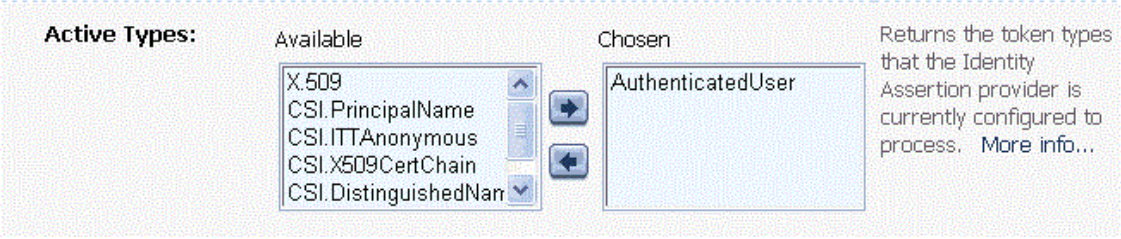
Example 4-1 PerimeterIdentityAsserterTokenTypes.java

```
package sample.security.providers.authentication.perimeterATN;
public class PerimeterIdentityAsserterTokenTypes
{
   public final static String TEST1_TYPE = 'Test 1";
   public final static String TEST2_TYPE = 'Test 2";
   public final static String TEST3_TYPE = 'Test 3";
}
```

How to Make New Token Types Available for Identity Assertion Provider Configurations

When you or an administrator configure a custom identity assertion provider (see Configure the Custom Identity Assertion Provider), the Supported Types field displays a list of the token types that the identity assertion provider supports. You enter one of the supported types in the Active Types field, as shown in Figure 4-1.

Figure 4-1 Configuring the Sample Identity Assertion Provider



The content for the Supported Types field is obtained from the `SupportedTypes` attribute of the MBean Definition File (MDF), which you use to generate your custom identity assertion provider's MBean type. An example from the sample identity assertion provider is shown in Example 4-2. (See Generate an MBean Type Using the WebLogic MBeanMaker.)

Example 4-2 SampleIdentityAsserter MDF: SupportedTypes Attribute

```
<MBeanType>
...
   <MBeanAttribute
    Name = "SupportedTypes"
    Type = "java.lang.String[]"
    Writeable = "false"
    Default = "new String[] {&quot;SamplePerimeterAtnToken&quot;}"
   />
...
</MBeanType>
```

Similarly, the content for the Active Types field is obtained from the `ActiveTypes` attribute of the MBean Definition File (MDF). You or an administrator can default the `ActiveTypes` attribute in the MDF so that it does not have to be set manually with the WebLogic Remote Console. An example from the sample identity assertion provider is shown in Example 4-3.

Example 4-3 SampleIdentityAsserter MDF: ActiveTypes Attribute with Default

```
<MBeanAttribute
 Name= "ActiveTypes"
 Type= "java.lang.String[]"
 Default = "new String[] { &quot;SamplePerimeterAtnToken&quot; }"
/>
```

While defaulting the `ActiveTypes` attribute is convenient, you should only do this if no other identity assertion provider will ever validate that token type. Otherwise, it would be easy to configure an invalid security realm (where more than one identity assertion provider attempts to validate the same token type). Best practice dictates that all MDFs for identity assertion providers turn off the token type by default; then an administrator can manually make the token type active by configuring the identity assertion provider that validates it.

> **Note**
>
> If an identity assertion provider is not developed and configured to validate and accept a token type, the authentication process will fail. For more information about configuring an identity assertion provider, see Configure the Custom Identity Assertion Provider .

Passing Tokens for Perimeter Authentication

An identity assertion provider can pass tokens from Java clients to servlets for the purpose of perimeter authentication. Tokens can be passed using HTTP headers, cookies, SSL certificates, or other mechanisms. For example, a string that is base 64-encoded (which enables the sending of binary data) can be sent to a servlet through an HTTP header. The value of this string can be a username, or some other string representation of a user's identity. The identity assertion provider used for perimeter authentication can then take that string and extract the username.

If the token is passed through HTTP headers or cookies, the token is equal to the header or cookie name, and the resource container passes the token to the part of the WebLogic Security Framework that handles authentication. The WebLogic Security Framework then passes the token to the identity assertion provider, unchanged.

WebLogic Server is designed to extend the single sign-on concept all the way to the perimeter through support for identity assertion. Identity assertion allows WebLogic Server to use the authentication mechanism provided by perimeter authentication schemes such as the Security Assertion Markup Language (SAML), the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO), or enhancements to protocols such as Common Secure Interoperability (CSI) v2 to achieve this functionality.

Common Secure Interoperability Version 2 (CSIv2)

WebLogic Server provides support for an Enterprise JavaBean (EJB) interoperability protocol based on Internet Inter-ORB (IIOP) (GIOP version 1.2) and the CORBA Common Secure Interoperability version 2 (CSIv2) specification. CSIv2 support in WebLogic Server:

- Interoperates with the Java Platform, Enterprise Edition (Java EE) reference implementation.
- Allows WebLogic Server IIOP clients to specify a username and password in the same manner as T3 clients.
- Supports Generic Security Services Application Programming Interface (GSSAPI) initial context tokens. For this release, only usernames and passwords and GSSUP (Generic Security Services Username Password) tokens are supported.

> **Note**
>
> The CSIv2 implementation in WebLogic Server passed Java EE Compatibility Test Suite (CTS) conformance testing.

The external interface to the CSIv2 implementation is a JAAS LoginModule that retrieves the username and password of the CORBA object. The JAAS LoginModule can be used in a WebLogic Java client or in a WebLogic Server instance that acts as a client to another Java EE application server. The JAAS LoginModule for the CSIv2 support is called `UsernamePasswordLoginModule`, and is located in the `weblogic.security.auth.login` package.

CSIv2 works in the following manner:

1. When creating a Security Extensions to Interoperable Object Reference (IOR), WebLogic Server adds a tagged component identifying the security mechanisms that the CORBA object supports. This tagged component includes transport information, client authentication information, and identity token/authorization token information.
2. The client evaluates the security mechanisms in the IOR and selects the mechanism that supports the options required by the server.
3. The client uses the SAS protocol to establish a security context with WebLogic Server. The SAS protocol defines messages contained within the service context of requests and replies. A context can be stateful or stateless.

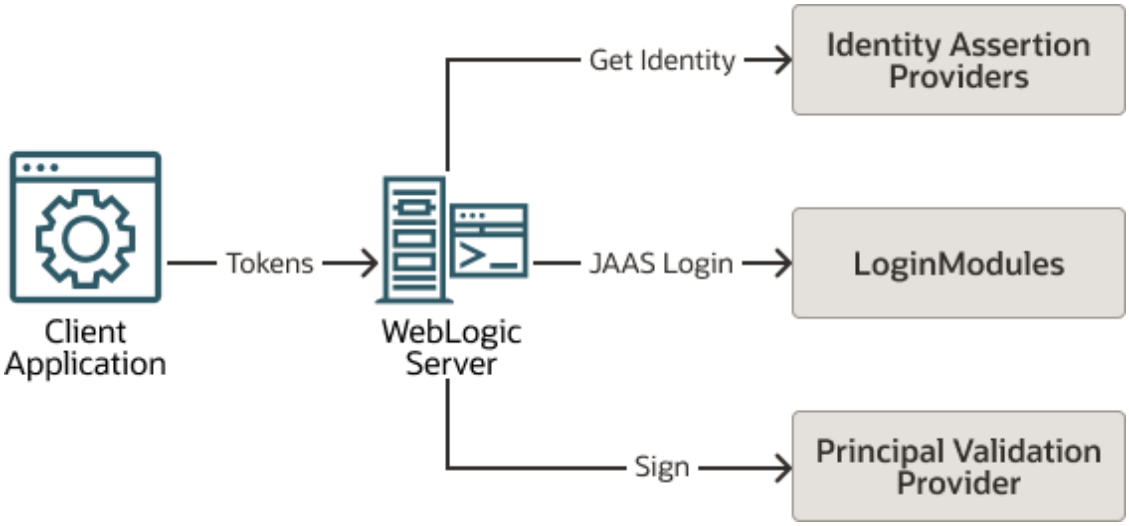
For information about using CSIv2, see Common Secure Interoperability Version 2 in *Understanding Security for Oracle WebLogic Server*. See LoginModules.

The Identity Assertion Process

In **perimeter authentication**, a system *outside* of WebLogic Server establishes trust via tokens (as opposed to the type of authentication described in The Authentication Process, where WebLogic Server establishes trust via usernames and passwords). Identity assertion providers are used as part of perimeter authentication process, which works as follows (see Figure 4-2):

1. A token from outside of WebLogic Server is passed to an identity assertion provider that is responsible for validating tokens of that type and that is configured as `active`.
2. If the token is successfully validated, the identity assertion provider maps the token to a WebLogic Server username, and sends that username back to WebLogic Server, which then continues the authentication process as described in The Authentication Process. Specifically, the username is sent via a Java Authentication and Authorization Service (JAAS) `CallbackHandler` and passed to each configured authentication provider's LoginModule, so that the LoginModule can populate the subject with the appropriate principals.

Figure 4-2 Perimeter Authentication



As Figure 4-2 also shows, perimeter authentication requires the same components as the authentication process described in The Authentication Process, but also adds an identity assertion provider.

Do You Need to Develop a Custom Identity Assertion Provider?

The WebLogic Identity Assertion providers support certificate authentication using X509 certificates, SPNEGO tokens, SAML assertion tokens, and CORBA Common Secure Interoperability version 2 (CSIv2) identity assertion.

The LDAP X509 Identity Assertion provider receives an X509 certificate, looks up the LDAP object for the user associated with that certificate, ensures that the certificate in the LDAP object matches the presented certificate, and then retrieves the name of the user from the LDAP object for the purpose of authentication.

The Negotiate Identity Assertion provider is used for SSO with Microsoft clients that support the SPNEGO protocol. The Negotiate Identity Assertion provider decodes SPNEGO tokens to obtain Kerberos tokens, validates the Kerberos tokens, and maps Kerberos tokens to WebLogic users. The Negotiate Identity Assertion provider utilizes the Java Generic Security Service (GSS) Application Programming Interface (API) to accept the GSS security context via Kerberos. The Negotiate Identity Assertion provider is for Windows NT Integrated Login.

The SAML Identity Assertion provider handles SAML assertion tokens when WebLogic Server acts as a SAML Service Provider site. The SAML Identity Assertion provider consumes and validates SAML assertion tokens and determines if the assertion is to be trusted (using either the proof material available in the SOAP message, the client certificate, or some other configuration indicator).

The default WebLogic Identity Assertion provider validates the token type, then maps X509 digital certificates and X501 distinguished names to WebLogic usernames. It also specifies a list of trusted client principals to use for CSIv2 identity assertion. The wildcard character (*) can be used to specify that all principals are trusted. If a client is not listed as a trusted client principal, the CSIv2 identity assertion fails and the invoke is rejected.

> **Note**
>
> To use the WebLogic Identity Assertion provider for X.501 and X.509 certificates, you have the option of using the default user name mapper that is supplied with the WebLogic Server product (`weblogic.security.providers.authentication.DefaultUserNameMapperImpl`) or providing you own implementation of the `weblogic.security.providers.authentication.UserNameMapper` interface.
>
> This interface maps a X.509 certificate to a WebLogic Server user name according to whatever scheme is appropriate for your needs. You can also use this interface to map from an X.501 distinguished name to a user name. You specify your implementation of this interface when you use the WebLogic Remote Console to configure an identity assertion provider.

The WebLogic Identity Assertion providers support the following token types:

- `AU_TYPE`, for a WebLogic `AuthenticatedUser` used as a token.
- `X509_TYPE`, for an X509 client certificate used as a token.
- `CSI_PRINCIPAL_TYPE`, for a CSIv2 principal name identity used as a token.
- `CSI_ANONYMOUS_TYPE`, for a CSIv2 anonymous identity used as a token.
- `CSI_X509_CERTCHAIN_TYPE`, for a CSIv2 X509 certificate chain identity used as a token.
- `CSI_DISTINGUISHED_NAME_TYPE`, for a CSIv2 distinguished name identity used as a token.
- `AUTHORIZATION_NEGOTIATE`, for a SPNEGO internal token used as a token.
- `SAML2_ASSERTION_DOM_TYPE`, for a SAML2 DOM element used as a token.
- `SAML2_ASSERTION_TYPE`, for a SAML2 string XML form used as a token.
- `SAML_SSO_CREDENTIAL_TYPE`, for a SAML string consisting of the TARGET parameter concatenated with the assertion itself and used as a token.
- `WSSE_PASSWORD_DIGEST_TYPE`, for a username token with a password type of password digest used as a token.
- `WWW_AUTHENTICATE_NEGOTIATE`, for a SPNEGO internal token used as a token.

If you want to perform additional identity assertion tasks or create new token types, then you need to develop a custom identity assertion provider.

How to Develop a Custom Identity Assertion Provider

If the WebLogic Identity Assertion provider does not meet your needs, you can develop a custom identity assertion provider by following these steps:

1. Create Runtime Classes Using the Appropriate SSPIs
2. Generate an MBean type for your custom identity assertion provider by completing the steps described in Generate an MBean Type Using the WebLogic MBeanMaker.
3. Configure the Custom Identity Assertion Provider
4. Consider whether you need to implement challenge identity assertion, as described in Challenge Identity Assertion.

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom identity assertion provider by following these steps:

- Implement the AuthenticationProviderV2 SSPI
- Implement the IdentityAsserterV2 SSPI

> **Note**
>
> If you want to create a separate LoginModule for your custom identity assertion provider (that is, not use the LoginModule from your authentication provider), you also need to implement the JAAS `LoginModule` interface, as described in Implement the JAAS LoginModule Interface.

For an example of how to create a runtime class for a custom identity assertion provider, see Example: Creating the Runtime Class for the Sample Identity Assertion Provider .

Implement the AuthenticationProviderV2 SSPI

> **Note**
>
> The `AuthenticationProvider` SSPI is deprecated in this release of WebLogic Server. Use the `AuthenticationProviderV2` SSPI instead.

To implement the `AuthenticationProviderV2` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following methods:

- getLoginModuleConfiguration

  ```
  public AppConfigurationEntry getLoginModuleConfiguration()
  ```

  The `getLoginModuleConfiguration` method obtains information about the authentication provider's associated LoginModule, which is returned as an `AppConfigurationEntry`. The `AppConfigurationEntry` is a Java Authentication and Authorization Service (JAAS) class that contains the classname of the LoginModule; the LoginModule's control flag (which was passed in via the authentication provider's associated MBean); and a configuration options map for the LoginModule (which allows other configuration information to be passed into the LoginModule).

  For more information about the `AppConfigurationEntry` class (located in the `javax.security.auth.login` package) and the control flag options for LoginModules, see the `AppConfigurationEntry` class (`http://docs.oracle.com/javase/8/docs/api/javax/security/auth/login/AppConfigurationEntry.html`) and the `Configuration` class (`http://docs.oracle.com/javase/8/docs/api/javax/security/auth/login/`

Configuration.html). For more information about LoginModules, see LoginModules. For more information about security providers and MBeans, see Understand Why You Need an MBean Type .

- getAssertionModuleConfiguration

```
public AppConfigurationEntry
getAssertionModuleConfiguration()
```

The `getAssertionModuleConfiguration` method obtains information about an identity assertion provider's associated LoginModule, which is returned as an `AppConfigurationEntry`. The `AppConfigurationEntry` is a JAAS class that contains the classname of the LoginModule; the LoginModule's control flag (which was passed in via the identity assertion provider's associated MBean); and a configuration options map for the LoginModule (which allows other configuration information to be passed into the LoginModule).

The LoginModules in this configuration must populate the Subject with required Principals, such as those of type WLSGroup, and must trust that the user has submitted sufficient proof to login and not require a password or some other proof material.

> **Note**
>
> The `assertIdentity()` method of an identity assertion provider is called every time identity assertion occurs, but the LoginModules may not be called if the Subject is cached. The `-Dweblogic.security.identityAssertionTTL` flag can be used to affect this behavior (for example, to modify the default TTL of 5 minutes or to disable the cache by setting the flag to -1).
>
> It is the responsibility of the identity assertion provider to ensure not just that the token is valid, but also that the user is still valid (for example, the user has not been deleted).

- getPrincipalValidator

```
public PrincipalValidator getPrincipalValidator()
```

The `getPrincipalValidator` method obtains a reference to the principal validation provider's runtime class (that is, the `PrincipalValidator` SSPI implementation). For more information, see Principal Validation Providers.

- getIdentityAsserter

```
public IdentityAsserterV2 getIdentityAsserter()
```

The `getIdentityAsserter` method obtains a reference to the identity assertion provider's runtime class (that is, the `IdentityAsserterV2` SSPI implementation). For more information, see Implement the IdentityAsserterV2 SSPI.

> **Note**
>
> When the LoginModule used for the identity assertion provider is the same as that used for an existing authentication provider, implementations for the methods in the `AuthenticationProviderV2` SSPI (excluding the `getIdentityAsserter` method) for identity assertion providers can just return `null`. An example of this is shown in Example 4-4.

See *Java API Reference for Oracle WebLogic Server* to know more about the `AuthenticationProviderV2` SSPI and the methods described above.

Implement the IdentityAsserterV2 SSPI

> **Note**
>
> The `IdentityAsserterV2` SSPI includes additional token types and a `handler` parameter to the `assertIdentity` method that can optionally be used to obtain additional information when asserting the identity. Although the `IdentityAsserter` SSPI is still supported, you should consider using the `IdentityAsserterV2` SSPI instead.

To implement the `IdentityAsserterV2` SSPI, provide implementations for the following method:

- assertIdentity

  ```
  public CallbackHandler assertIdentity(String type, Object token,
  ContextHandler handler) throws IdentityAssertionException;
  ```

  The `assertIdentity` method asserts an identity based on the token identity information that is supplied. In other words, the purpose of this method is to validate any tokens that are not currently trusted against trusted client principals. The `type` parameter represents the token type to be used for the identity assertion.

  > **Note**
  >
  > HTTP header names are considered case *insensitive*. However, HTTP/2 converts HTTP header names to lowercase when sending requests. Make sure that your Identity Assertion provider uses a case insensitive comparison for the token types passed to the `assertIdentity()` method when using HTTP headers to pass tokens.

  The `token` parameter contains the actual identity information. The `handler` parameter is a `ContextHandler` object that can optionally be used to obtain additional information that may be used in asserting the identity. The `CallbackHandler` returned from the `assertIdentity` method is passed to all configured authentication providers' LoginModules to perform principal mapping, and should contain the asserted username. If the `CallbackHandler` is `null`, this signifies that the anonymous user should be used.

  A `CallbackHandler` is a highly-flexible JAAS standard that allows a variable number of arguments to be passed as complex objects to a method. For more information about `CallbackHandlers`, see the `CallbackHandler` interface (`http://docs.oracle.com/javase/8/docs/api/javax/security/auth/callback/CallbackHandler.html`).

> **Note**
>
> The `assertIdentity()` method of an identity assertion provider is called every time identity assertion occurs, but the LoginModules may not be called if the Subject is cached. The `-Dweblogic.security.identityAssertionTTL` flag can be used to affect this behavior (for example, to modify the default TTL of 5 minutes or to disable the cache by setting the flag to -1).
>
> It is the responsibility of the identity assertion provider to ensure not just that the token is valid, but also that the user is still valid (for example, the user has not been deleted).

See *Java API Reference for Oracle WebLogic Server* to know more about the IdentityAsserterV2 SSPI and the methods described above.

Example: Creating the Runtime Class for the Sample Identity Assertion Provider

Example 4-4 shows the `SampleIdentityAsserterProviderImpl.java` class, which is the runtime class for the sample identity assertion provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription`, and `shutdown` (as described in Understand the Purpose of the Provider SSPIs.)
- The four methods in the `AuthenticationProviderV2` SSPI: the `getLoginModuleConfiguration`, `getAssertionModuleConfiguration`, `getPrincipalValidator`, and `getIdentityAsserter` methods (as described in Implement the AuthenticationProviderV2 SSPI.
- The method in the `IdentityAsserterV2` SSPI: the `assertIdentity` method (described in Implement the IdentityAsserterV2 SSPI).

> **Note**
>
> The bold face code in Example 4-4 highlights the class declaration and the method signatures.

Example 4-4 SampleIdentityAsserterProviderImpl.java

```
package examples.security.providers.identityassertion.simple;
import jakarta.security.auth.callback.CallbackHandler;
import jakarta.security.auth.login.AppConfigurationEntry;
import weblogic.management.security.ProviderMBean;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AuthenticationProviderV2;
import weblogic.security.spi.IdentityAsserterV2;
import weblogic.security.spi.IdentityAssertionException;
import weblogic.security.spi.PrincipalValidator;
import weblogic.security.spi.SecurityServices;
public final class SimpleSampleIdentityAsserterProviderImpl implements AuthenticationProviderV2,
IdentityAsserterV2
{
   final static private String TOKEN_TYPE   = "SamplePerimeterAtnToken";
   final static private String TOKEN_PREFIX = "username=";
   private String description;
   public void initialize(ProviderMBean mbean, SecurityServices services)
```

```
{
   System.out.println("SimpleSampleIdentityAsserterProviderImpl.initialize");
   SimpleSampleIdentityAsserterMBean myMBean = (SimpleSampleIdentityAsserterMBean)mbean;
   description = myMBean.getDescription() + "\n" + myMBean.getVersion();
}
public String getDescription()
{
   return description;
}
public void shutdown()
{
   System.out.println("SimpleSampleIdentityAsserterProviderImpl.shutdown");
}
public IdentityAsserterV2 getIdentityAsserter()
{
   return this;
}
public CallbackHandler assertIdentity(String type, Object token, ContextHandler context) throws
IdentityAssertionException
{
   System.out.println("SimpleSampleIdentityAsserterProviderImpl.assertIdentity");
   System.out.println("\tType\t\t= "  + type);
   System.out.println("\tToken\t\t= " + token);
   if (!(TOKEN_TYPE.equals(type))) {
      String error = "SimpleSampleIdentityAsserter received unknown token type \""
         + type + "\"." + " Expected " + TOKEN_TYPE;
      System.out.println("\tError: " + error);
      throw new IdentityAssertionException(error);
   }
   if (!(token instanceof byte[])) {
      String error = "SimpleSampleIdentityAsserter received unknown token class \""
         + token.getClass() + "\"." + " Expected a byte[].";
      System.out.println("\tError: " + error);
      throw new IdentityAssertionException(error);
   }
   byte[] tokenBytes = (byte[])token;
   if (tokenBytes == null || tokenBytes.length < 1) {
      String error = "SimpleSampleIdentityAsserter received empty token byte array";
      System.out.println("\tError: " + error);
      throw new IdentityAssertionException(error);
   }
   String tokenStr = new String(tokenBytes);
   if (!(tokenStr.startsWith(TOKEN_PREFIX))) {
      String error = "SimpleSampleIdentityAsserter received unknown token string \""
         + type + "\"." + " Expected " + TOKEN_PREFIX + "username";
      System.out.println("\tError: " + error);
      throw new IdentityAssertionException(error);
   }
   String userName = tokenStr.substring(TOKEN_PREFIX.length());
   System.out.println("\tuserName\t= " + userName);
   return new SimpleSampleCallbackHandlerImpl(userName);
}
public AppConfigurationEntry getLoginModuleConfiguration()
{
   return null;
}
public AppConfigurationEntry getAssertionModuleConfiguration()
{
   return null;
}
public PrincipalValidator getPrincipalValidator()
{
```

```
      return null;
   }
}
```

Example 4-5 shows the sample `CallbackHandler` implementation that is used along with the `SampleIdentityAsserterProviderImpl.java` runtime class. This `CallbackHandler` implementation is used to send the username back to an authentication provider's LoginModule.

Example 4-5 SampleCallbackHandlerImpl.java

```
package examples.security.providers.identityassertion.simple;
import jakarta.security.auth.callback.Callback;
import jakarta.security.auth.callback.NameCallback;
import jakarta.security.auth.callback.CallbackHandler;
import jakarta.security.auth.callback.UnsupportedCallbackException;
/*package*/ class SimpleSimpleSampleCallbackHandler implements CallbackHandler
{
   private String userName;
   /*package*/ SimpleSampleCallbackHandlerImpl(String user)
   {
      userName = user;
   }
   public void handle(Callback[] callbacks) throws UnsupportedCallbackException
   {
      for (int i = 0; i < callbacks.length; i++) {
            Callback callback = callbacks[i];
            if (!(callback instanceof NameCallback)) {
               throw new UnsupportedCallbackException(callback, "Unrecognized
                  Callback");
            }
            NameCallback nameCallback = (NameCallback)callback;
            nameCallback.setName(userName);
      }
   }
}
```

Configure the Custom Identity Assertion Provider

Configuring a custom identity assertion provider means that you are adding the custom identity assertion provider to your security realm, where it can be accessed by applications requiring identity assertion services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers.

> **Note**
>
> The steps for configuring a custom identity assertion provider are described in Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

Challenge Identity Assertion

The Challenge Identity Asserter interface supports challenge response schemes in which multiple challenges, responses messages, and state are required. The Challenge Identity Asserter interface allows identity assertion providers to support authentication protocols such

as Microsoft's Windows NT Challenge/Response (NTLM), Simple and Protected GSS-API Negotiation Mechanism (SPNEGO), and other challenge/response authentication mechanisms.

Challenge/Response Limitations in the Java Servlet API 2.3 Environment

The WebLogic Security Framework allows you to provide a custom authentication and identity assertion provider. However, due to the nature of the Java Servlet API 2.3 specification, the interaction between the authentication provider and the client or other servers is architecturally limited during the authentication process. This restricts authentication mechanisms to those that are compatible with the authentication mechanisms the Servlet container offers: basic, form, and certificate.

Servlet authentication filters, which are described in Servlet Authentication Filters have fewer architecturally-dependence limitations; that is, they are not dependent on the authentication mechanisms offered by the servlet container. By allowing filters to be invoked prior to the container beginning the authentication process, a security realm can implement a wider scope of authentication mechanisms. For example, a servlet authentication filter could redirect the user to a SAML provider site for authentication.

Servlet authentication filters provide a convenient way to implement a challenge/response protocol in your environment. Filters allow your Challenge Identity Assertion interface to loop through your challenge/response mechanism as often as needed to complete the challenge.

Filters and The Role of the weblogic.security.services.Authentication Class

Servlet authentication filters allow you to implement a challenge/response protocol without being limited to the authentication mechanisms compatible with the Servlet container. However, because servlet authentication filters operate outside of the authentication environment provided by the Security Framework, they cannot depend on the Security Framework to determine provider context, and require an API to drive the multiple-challenge identity assertion process.

The `weblogic.security.services.Authentication` class has been extended to allow multiple challenge/response identity assertion from a servlet authentication filter. The methods and interface provide a wrapper for the `ChallengeIdentityAsserterV2` and `ProviderChallengeContext` interfaces so that you can invoke them from a servlet authentication filter.

There is no other documented way to perform a multiple challenge/response dialog from a servlet authentication filter within the context of the Security Framework. Your servlet authentication filter cannot directly invoke the `ChallengeIdentityAsserterV2` and `ProviderChallengeContext` interfaces.

Therefore, you need to implement the `ChallengeIdentityAsserterV2` and `ProviderChallengeContext` interfaces, and then use the `weblogic.security.services.Authentication` methods and `AppChallengeContext` interface to invoke them from a servlet authentication filter.

How to Develop a Challenge Identity Asserter

To develop a Challenge Identity Asserter:

- Implement the AuthenticationProviderV2 SSPI
- Implement the IdentityAsserterV2 SSPI
- Implement the ChallengeIdentityAsserterV2 Interface

- Invoke the weblogic.security.services Challenge Identity Methods
- Invoke the weblogic.security.services AppChallengeContext Methods

Implement the ChallengeIdentityAsserterV2 Interface

The `ChallengeIdentityAsserterV2` interface extends the `IdentityAsserterV2` SSPI. You must implement the `ChallengeIdentityAsserterV2` interface in addition to the `IdentityAsserterV2` SSPI.

Provide an implementation for all of the `IdentityAsserterV2` methods, *and* the following methods:

- assertChallengeIdentity

  ```
  ProviderChallengeContext assertChallengeIdentity(String tokenType, Object token,
  ContextHandler handler)
  ```

  Use the supplied client token to establish client identity, possibly with multiple challenges. This method returns your implementation of the `ProviderChallengeContext` interface. The `ProviderChallengeContext` interface provides a means to query the state of the challenges.

- continueChallengeIdentity

  ```
  void continueChallengeIdentity(ProviderChallengeContext context, String tokenType,
  Object token,
  ContextHandler handler)
  ```

  Use the supplied provider context and client token to continue establishing client identity.

- getChallengeToken

  ```
  Object getChallengeToken(String type, ContextHandler handler)
  ```

  This method returns the identity assertion provider's challenge token.

Implement the ProviderChallengeContext Interface

The `ProviderChallengeContext` interface provides a means to query the state of the challenges. It allows the `assertChallengeIdentity` and `continueChallengeIdentity` methods of the `ChallengeIdentityAsserterV2` interface to return either the callback handler or a new challenge to which the client must respond.

To implement the `ProviderChallengeContext` interface, provide implementations for the following methods:

- getCallbackHandler

  ```
  CallbackHandler getCallbackHandler()
  ```

  This method returns the callback handler for the challenge identity assertion. Call this method only when the `hasChallengeIdentityCompleted` method returns true.

- getChallengeToken

  ```
  Object getChallengeToken()
  ```

  This method returns the challenge token for the challenge identity assertion. Call this method only when the `hasChallengeIdentityCompleted` method returns false.

- hasChallengeIdentityCompleted

```
boolean hasChallengeIdentityCompleted
```

This method returns whether the challenge identity assertion has completed. It returns true if the challenge identity assertion has completed, false if not. If true, the caller should use the `getCallbackHandler` method. If false, then the caller should use the `getChallengeToken` method.

Invoke the weblogic.security.services Challenge Identity Methods

Have your servlet authentication filter invoke the following `weblogic.security.services.Authentication` methods instead of calling the `ChallengeIdentityAsserterV2` SSPI directly:

- assertChallengeIdentity

  ```
  AppChallengeContext assertChallengeIdentity(String tokenType, Object token,
  AppContext appContext)
  ```

  Use the supplied client token to establish client identity, possibly with multiple challenges. This method returns the context of the challenge identity assertion. This result may contain either the authenticated subject or an additional challenge to which the client must respond. The `AppChallengeContext` interface provides a means to query the state of the challenges.

- continueChallengeIdentity

  ```
  void continueChallengeIdentity(AppChallengeContext context, String tokenType,
   Object token, AppContext appContext)
  ```

  Use the supplied provider context and client token to continue establishing client identity.

- getChallengeToken

  ```
  Object getChallengeToken
  ```

  This method returns the initial challenge token for the challenge identity assertion.

Invoke the weblogic.security.services AppChallengeContext Methods

Have your servlet authentication filter invoke the following AppChallengeContext methods instead of invoking the `ProviderChallengeContext` interface directly:

- getAuthenticatedSubject

  ```
  Subject getAuthenticatedSubject()
  ```

  Returns the authenticated subject for the challenge identity assertion. Call this method only when the `hasChallengeIdentityCompleted` method returns true.

- getChallengeToken

  ```
  Object getChallengeToken()
  ```

  This method returns the challenge token for the challenge identity assertion. Call this method only when the `hasChallengeIdentityCompleted` method returns false.

- hasChallengeIdentityCompleted

  ```
  boolean hasChallengeIdentityCompleted()
  ```

  This method returns whether the challenge identity assertion has completed. It returns true if the challenge identity assertion has completed, false if not. If true, the caller should use

the `getCallbackHandler` method. If false, then the caller should use the `getChallengeToken` method.

Implementing Challenge Identity Assertion from a Filter

In the following code flow, assume that the servlet authentication filter, which is described in Servlet Authentication Filters handles the HTTP level interactions (Authorization and WWW-Authenticate) and is also responsible for calling the `weblogic.security.services.Authentication` methods and interfaces to drive the Challenge Identity Assertion process.

1. Browser sends a request
2. Filter sees requests and no authorization header, so it calls the `weblogic.security.services.Authentication` `getChallengeToken` method to get an initial token and sends a 401 response with a WWW-Authenticate negotiate header back
3. Browser sees 401 with WWW-Authenticate and responds with a new request and a Authorization Negotiate token.
   a. Filter sees this and calls the weblogic.security.services.Authentication `assertChallengeIdentity` method. `assertChallengeIdentity` takes the token as input, processes it according to whatever rules it needs to follow for the assertion process it is following (for example, if NTLM, then do whatever NTLM requires to process the token), and determine if that succeeded or not. `assertChallengeIdentity` returns your implementation of the `AppChallengeContext` interface.
   b. Filter calls appChallengeContext `hasChallengeCompleted` method. Use the AppChallengeContext `hasChallengeIdentityCompleted` method to see if the challenge has completed. For example, it can determine if the callback handler is not null, meaning that it contains a username, and return true. In this use it returns false, so it must issue another challenge to the client. The filter then calls AppChallengeContext `getChallengeToken` to get the token to challenge back with.
   c. Filter likely stores the AppChallengeContext somewhere such as a session attribute.
   d. Filter sends a 401 response with an WWW-Authenticate negotiate and the new token.
4. Browser sees the new challenge and responds again with an authorization header.
   a. Filter sees this and calls the weblogic.security.services.Authentication `continueChallengeIdentity` method.
   b. Filter calls the AppChallengeContext `hasChallengeCompleted` method. If it returns false another challenge is in order, so call the AppChallengeContext getChallengeToken method to get the token to challenge back with, and so forth. If it returned true, then the challenge has completed and the filter would then call AppChallengeContext `getAuthenticatedSubject` method and perform a runAs(subject, request).

5

Principal Validation Providers

This chapter describes principal validation provider concepts and functionality, and provides step-by-step instructions for developing a custom principal validation provider. Authentication providers rely on principal validation providers to sign and verify the authenticity of principals (users and groups) contained within a subject. Such verification provides an additional level of trust and may reduce the likelihood of malicious principal tampering. Verification of the subject's principals takes place during the WebLogic Server's demarshalling of RMI client requests for each invocation. The authenticity of the subject's principals is also verified when making authorization decisions.

This chapter includes the following sections:

Principal Validation Concepts

Before you develop a principal validation provider, you need to understand the following concepts:

Principal Validation and Principal Types

Like identity assertion providers support specific types of tokens, principal validation providers support specific types of principals. For example, the WebLogic Principal Validation provider (described in Do You Need to Develop a Custom Principal Validation Provider?) signs and verifies the authenticity of WebLogic Server principals.

The principal validation provider that is associated with the configured authentication provider (as described in How Principal Validation Providers Differ From Other Types of Security Providers) will sign and verify all the principals stored in the subject that are of the type the principal validation provider is designed to support.

How Principal Validation Providers Differ From Other Types of Security Providers

A principal validation provider is a special type of security provider that primarily acts as a helper to an authentication provider. The main function of a principal validation provider is to prevent malicious individuals from tampering with the principals stored in a subject.

The `AuthenticationProvider` SSPI (as described in Implement the AuthenticationProviderV2 SSPI) includes a method called `getPrincipalValidator`. In this method, you specify the principal validation provider's runtime class to be used with the authentication provider. The principal validation provider's runtime class can be the one Oracle provides (called the WebLogic Principal Validation provider) or one you develop (called a custom principal validation provider). An example of using the WebLogic Principal Validation provider in an authentication provider's `getPrincipalValidator` method is shown in Figure 3-1.

You do not generate MBean types for principal validation providers. Instead, specify a principal validator from the authentication provider.

Security Exceptions Resulting from Invalid Principals

When the WebLogic Security Framework attempts an authentication (or authorization) operation, it checks the subject's principals to see if they are valid. If a principal is not valid, the WebLogic Security Framework throws a security exception with text indicating that the subject is invalid. A subject may be invalid because:

- A principal in the subject does not have a corresponding principal validation provider configured (which means there is no way for the WebLogic Security Framework to validate the subject).

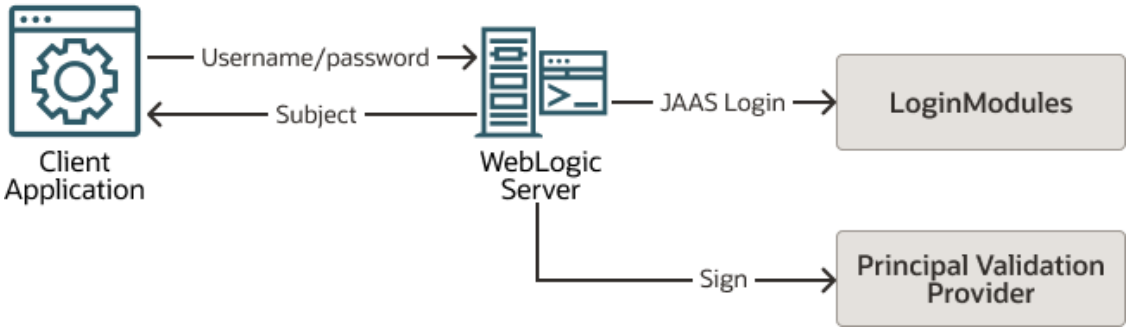
  > **Note**
  >
  > Because you can have multiple principals in a subject, each stored by the LoginModule of a different authentication provider, the principals can have different principal validation providers.

- A principal was signed in another WebLogic Server security domain (with a different credential from this security domain) and the caller is trying to use it in the current domain.
- A principal with an invalid signature was created as part of an attempt to compromise security.
- A subject never had its principals signed.

The Principal Validation Process

As shown in Figure 5-1, a user attempts to log into a system using a username/password combination. WebLogic Server establishes trust by calling the configured authentication provider's LoginModule, which validates the user's username and password and returns a subject that is populated with principals per Java Authentication and Authorization Service (JAAS) requirements.

Figure 5-1 The Principal Validation Process



WebLogic Server passes the subject to the specified principal validation provider, which signs the principals and then returns them to the client application via WebLogic Server. Whenever the principals stored within the subject are required for other security operations, the same principal validation provider will verify that the principals stored within the subject have not been modified since they were signed.

Do You Need to Develop a Custom Principal Validation Provider?

The default (that is, active) security realm for WebLogic Server includes a WebLogic Principal Validation provider. Much like an identity assertion provider supports a specific type of token, a principal validation provider signs and verifies the authenticity of a specific type of principal. The WebLogic Principal Validation provider signs and verifies WebLogic Server principals. In other words, it signs and verifies principals that represent WebLogic Server users or WebLogic Server groups.

> **Note**
>
> You can use the `WLSPrincipals` class (located in the `weblogic.security.principal` package) to determine whether a principal (user or group) has special meaning to WebLogic Server. (That is, whether it is a predefined WebLogic Server user or WebLogic Server group.) Furthermore, any principal that is going to represent a WebLogic Server user or group needs to implement the `WLSUser` and `WLSGroup` interfaces (available in the `weblogic.security.spi` package).
>
> `WLSPrincipals` is used only by `PrincipalValidatorImpl`, not by the Security Framework. An authentication provider can implement its own principal validator, or it can use the `PrincipalValidatorImpl`. If you configure an authentication provider with custom principal validators, then the `WLSPrincipals` interface is not used.
>
> An authentication provider needs to implement the `WLSPrincipals` interface if the provider is going to use `PrincipalValidatorImpl`.

The WebLogic Principal Validation provider includes implementations of the `WLSUser` and `WLSGroup` interfaces, named `WLSUserImpl` and `WLSGroupImpl`. These are located in the `weblogic.security.principal` package.

It also includes an implementation of the `PrincipalValidator` SSPI called `PrincipalValidatorImpl`, located in the `weblogic.security.provider` package. The `sign()`

method in the `PrincipalValidatorImpl` class generates a random seed and computes a digest based on that random seed. (See Implement the PrincipalValidator SSPI.)

How to Use the WebLogic Principal Validation Provider

If you have simple user and group principals (that is, they only have a name), and you want to use the WebLogic Principal Validation provider:

- Use the existing `weblogic.security.principal.WLSUserImpl` and `weblogic.security.principal.WLSGroupImpl` classes. See the `WLSUser` and `WLSGroup` interfaces in the `weblogic.security.spi` package for usage information.
- Use the `weblogic.security.provider.PrincipalValidatorImpl` class. See the `PrincipalValidator` SSPI for usage information.

If you have user or group principals with extra data members (that is, in addition to a name), and you want to use the WebLogic Principal Validation provider:

- Write your own `UserImpl` and `GroupImpl` classes.
- Extend the `weblogic.security.principal.WLSAbstractPrincipal` class.
- Implement the `weblogic.security.spi.WLSUser` and `weblogic.security.spi.WLSGroup` interfaces.
- Implement the `equals()` method to include your extra data members. Your implementation should call the `super.equals()` method when complete so the `WLSAbstractPrincipal` can validate the remaining data.

> **Note**
>
> By default, only the user or group name will be validated. If you want to validate your extra data members as well, then implement the `getSignedData()` method.

- Use the `weblogic.security.provider.PrincipalValidatorImpl` class. See the `PrincipalValidator` SSPI for usage information.

If you have your own validation scheme and do not want to use the WebLogic Principal Validation provider, or if you want to provide validation for principals other than WebLogic Server principals, then you need to develop a custom principal validation provider.

How to Develop a Custom Principal Validation Provider

To develop a custom principal validation provider:

- Write your own `UserImpl` and `GroupImpl` classes by:
  - Implementing the weblogic.security.spi.WLSUser and weblogic.security.spi.WLSGroup interfaces.
  - Implementing the java.io.Serializable interfaces.
- Write your own `PrincipalValidationImpl` class by implementing the `weblogic.security.spi.PrincipalValidator` SSPI. (See Implement the PrincipalValidator SSPI.)

Implement the PrincipalValidator SSPI

To implement the `PrincipalValidator` SSPI, provide implementations for the following methods:

- validate

  ```
  public boolean validate(Principal principal) throws SecurityException;
  ```

  The `validate` method takes a principal as an argument and attempts to validate it. In other words, this method verifies that the principal was not altered since it was signed.

- sign

  ```
  public boolean sign(Principal principal);
  ```

  The `sign` method takes a principal as an argument and signs it to assure trust. This allows the principal to later be verified using the `validate` method.

  Your implementation of the `sign` method should be a secret algorithm that malicious individuals cannot easily recreate. You can include that algorithm within the `sign` method itself, have the `sign` method call out to a server for a token it should use to sign the principal, or implement some other way of signing the principal.

- getPrincipalBaseClass

  ```
  public Class getPrincipalBaseClass();
  ```

  The `getPrincipalBaseClass` method returns the base class of principals that this principal validation provider knows how to validate and sign.

See *Java API Reference for Oracle WebLogic Server* for the PrincipalValidator SSPI.

6

Authorization Providers

This chapter describes authorization provider concepts and functionality, and provides step-by-step instructions for developing a custom authorization provider.
Authorization is the process whereby the interactions between users and WebLogic resources are controlled, based on user identity or other information. In other words, authorization answers the question, "What can you access?" In WebLogic Server, an authorization provider is used to limit the interactions between users and WebLogic resources to ensure integrity, confidentiality, and availability.

This chapter includes the following sections:

Authorization Concepts

Before you develop an authorization provider, you need to understand the following concepts:

Access Decisions

Like LoginModules for authentication providers, an **Access Decision** is the component of an authorization provider that actually answers the "is access allowed?" question. Specifically, an Access Decision is asked whether a subject has permission to perform a given operation on a WebLogic resource, with specific parameters in an application. Given this information, the Access Decision responds with a result of `PERMIT`, `DENY`, or `ABSTAIN`.

> **Note**
>
> See Implement the AccessDecision SSPI.

Using the Jakarta Authorization Contract for Containers

The Jakarta Authorization Contract for Containers (JACC) is part of Jakarta EE. JACC extends the permission-based security model to EJBs and Servlets. JACC is defined by JSR-115 (`https://jakarta.ee/specifications/authorization/2.0/`).

JACC provides an alternate authorization mechanism for the EJB and Servlet containers in a WebLogic Server domain. When JACC is configured, the WebLogic Security framework access decisions, adjudication, and role mapping functions are not used for EJB and Servlet authorization decisions.

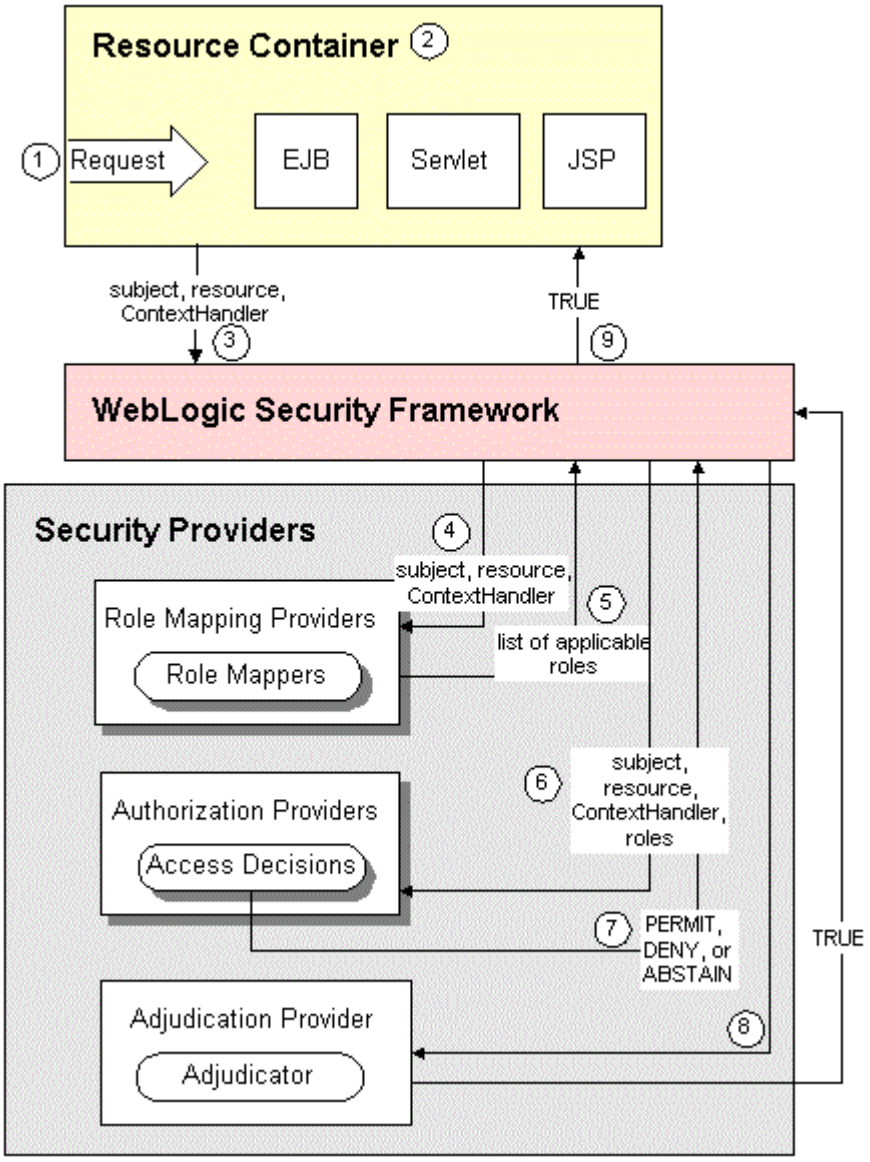
> **Note**
>
> You cannot use the JACC framework in conjunction with the WebLogic Security framework. The JACC classes used by WebLogic Server do not include an implementation of a Policy object for rendering decisions but instead rely on the jakarta.security.Policy (http://docs.oracle.com/javase/8/docs/api/java/security/Policy.html) object.

WebLogic Server implements a JACC provider that, although fully compliant with JSR-115, is not as optimized as the WebLogic Authentication provider. The Java JACC classes are used for rendering access decisions. Because JSR-115 does not define how to address role mapping, WebLogic JACC classes are used for role-to-principal mapping. For information on developing a JACC provider, see https://javaee.github.io/javaee-spec/javadocs/javax/security/jacc/package-summary.html.

The Authorization Process

Figure 6-1 illustrates how authorization providers (and the associated adjudication and role mapping providers) interact with the WebLogic Security Framework during the authorization process, and an explanation follows.

Figure 6-1 Authorization Providers and the Authorization Process



Generally, authorization is performed in the following manner:

1. A user or system process requests a WebLogic resource on which it will attempt to perform a given operation.
2. The resource container that handles the type of WebLogic resource being requested receives the request (for example, the EJB container receives the request for an EJB resource).

> **Note**
>
> The resource container could be the container that handles any one of the WebLogic Resources described in Security Providers and WebLogic Resources.

3. The resource container constructs a `ContextHandler` object that may be used by the configured role mapping providers and the configured authorization providers' Access Decisions to obtain information associated with the context of the request.

   > **Note**
   >
   > See ContextHandlers and WebLogic Resources, Access Decisions, and Role Mapping Providers.

   The resource container calls the WebLogic Security Framework, passing in the subject, the WebLogic resource, and optionally, the `ContextHandler` object (to provide additional input for the decision).

4. The WebLogic Security Framework calls the configured role mapping providers.
5. The role mapping providers use the `ContextHandler` to request various pieces of information about the request. They construct a set of `Callback` objects that represent the type of information being requested. This set of `Callback` objects is then passed as an array to the `ContextHandler` using the `handle` method.

   The role mapping providers use the values contained in the `Callback` objects, the subject, and the resource to compute a list of security roles to which the subject making the request is entitled, and pass the list of applicable security roles back to the WebLogic Security Framework.
6. The WebLogic Security Framework delegates the actual decision about whether the subject is entitled to perform the requested action on the WebLogic resource to the configured authorization providers.

   The authorization providers' Access Decisions also use the `ContextHandler` to request various pieces of information about the request. They too construct a set of `Callback` objects that represent the type of information being requested. This set of `Callback` objects is then passed as an array to the `ContextHandler` using the `handle` method. (The process is the same as described for role mapping providers in Step 5.)
7. The `isAccessAllowed` method of each configured authorization provider's Access Decision is called to determine if the subject is authorized to perform the requested access, based on the `ContextHandler`, subject, WebLogic resource, and security roles. Each `isAccessAllowed` method can return one of three values:

   - `PERMIT`, indicates that the requested access is permitted.
   - `DENY`, indicates that the requested access is explicitly denied.
   - `ABSTAIN`, indicates that the Access Decision was unable to render an explicit decision.

   This process continues until all Access Decisions are used.
8. The WebLogic Security Framework delegates the job of reconciling any discrepancies among the results rendered by the configured authorization providers' Access Decisions to the adjudication provider. The adjudication provider determines the ultimate outcome of the authorization decision.

   > **Note**
   >
   > See Adjudication Providers.

9. The adjudication provider returns either a `TRUE` or `FALSE` verdict, which is forwarded to the resource container through the WebLogic Security Framework.
   - If the decision is `TRUE`, the resource container dispatches the request to the protected WebLogic resource.
   - If the decision is `FALSE`, the resource container throws a security exception that indicates that the requestor was not authorized to perform the requested access on the protected WebLogic resource.

Do You Need to Develop a Custom Authorization Provider?

The default (that is, active) security realm for WebLogic Server includes the WebLogic Authorization provider and the XACML Authorization provider.

> **Note**
>
> The WebLogic Authorization provider, also referred to as the DefaultAuthorizer, is deprecated in WebLogic Server 14.1.1.0.0 and will be removed in a future release. Instead, the XACML Authorization provider is the default authorization provider.

The XACML Authorization provider returns an access decision using a policy-based authorization engine to determine if a particular user is allowed access to a protected WebLogic resource. The XACML Authorization provider also supports the deployment and undeployment of security policies within the system. If you want to use an authorization mechanism that already exists within your organization, you could create a custom authorization provider to tie into that system.

Does Your Custom Authorization Provider Need to Support Application Versioning?

All authorization, role mapping, and credential mapping providers for the security realm must support application versioning in order for an application to be deployed using versions. If you develop a custom security provider for authorization, role mapping, or credential mapping and need to support versioned applications, you must implement the Versionable Application SSPI, as described in Versionable Application Providers.

Is Your Custom Authorization Provider Thread Safe?

For the best performance, and by default, Weblogic Server supports parallel modification to security policy and roles during application and module deployment. For this reason, deployable authorization and role mapping providers configured in the security realm should support parallel calls. The WebLogic deployable XACML Authorization and Role Mapping providers meet this requirement.

However, custom deployable authorization and role mapping providers may or may not support parallel calls. If your custom deployable authorization or role mapping providers do **not** support parallel calls, you need to disable the parallel security policy and role modification and instead enforce a synchronization mechanism that results in each application and module being placed in a queue and deployed sequentially.

> **Note**
>
> Enabling the synchronization mechanism affects every deployable provider configured in the realm, including the predefined WebLogic Server providers. Enabling the synchronization mechanism may negatively impact the performance of these providers.

See *Administering Security for Oracle WebLogic Server* for information on how to turn on this synchronization enforcement mechanism.

How to Develop a Custom Authorization Provider

If the XACML Authorization provider does not meet your needs, you can develop a custom authorization provider by following these steps:

1. Create Runtime Classes Using the Appropriate SSPIs, or, optionally, implement the Bulk Authorization Providers
2. Optionally, implement the Policy Consumer SSPI
3. Optionally, implement the PolicyStoreMBean
4. Generate an MBean type for your custom authorization provider by completing the steps described in Generate an MBean Type Using the WebLogic MBeanMaker.
5. Configure a Custom Authorization Provider
6. Provide a Mechanism for Security Policy Management

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Determine Which Provider Interface You Will Implement
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom authorization provider by following these steps:

- Implement the AuthorizationProvider SSPI *or* Implement the DeployableAuthorizationProviderV2 SSPI
- Implement the AccessDecision SSPI

> **Note**
>
> At least one authorization provider in a security realm must implement the `DeployableAuthorizationProvider` SSPI, or else it will be impossible to deploy Web applications and EJBs.

For an example of how to create a runtime class for a custom authorization provider, see Example: Creating the Runtime Class for the Sample Authorization Provider .

Implement the AuthorizationProvider SSPI

To implement the `AuthorizationProvider` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following method:

- getAccessDecision

  ```
  public AccessDecision getAccessDecision();
  ```

  The `getAccessDecision` method obtains the implementation of the `AccessDecision` SSPI. For a single runtime class called `MyAuthorizationProviderImpl.java`, the implementation of the `getAccessDecision` method would be:

  ```
  return this;
  ```

  If there are two runtime classes, then the implementation of the `getAccessDecision` method could be:

  ```
  return new MyAccessDecisionImpl;
  ```

  This is because the runtime class that implements the `AuthorizationProvider` SSPI is used as a factory to obtain classes that implement the `AccessDecision` SSPI.

See *Java API Reference for Oracle WebLogic Server* for the AuthorizationProvider SSPI.

Implement the DeployableAuthorizationProviderV2 SSPI

To implement the `DeployableAuthorizationProviderV2` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs, Implement the AuthorizationProvider SSPI, *and* the following methods:

- deleteApplicationPolicies

  ```
  public void deleteApplicationPolicies(ApplicationInfo application) throws
  ResourceRemovalException
  ```

  The `deleteApplicationPolicies` method deletes all policies for an application. The `deleteApplicationPolicies` method is called only on the Administration Server.

- deployExcludedPolicy

  ```
  public void deleteApplicationPolicies(DeployPolicyHandle handle, Resource
   resource) throws ResourceCreationException
  ```

  The `deployExcludedPolicy` method deploys a policy that always denies access. If a policy already exists, it is removed and replaced by this policy.

- deployPolicy

  ```
  public void deployPolicy(DeployPolicyHandle handle, Resource resource,
   String[] roleNames) throws ResourceCreationException
  ```

  The `deployPolicy` method creates a security policy on behalf of a deployed Web application or EJB, based on the WebLogic resource to which the security policy should apply and the security role names that are in the security policy.

- deployUncheckedPolicy

  ```
  public void deployUncheckedPolicy(DeployPolicyHandle handle, Resource
   resource) throws ResourceCreationException
  ```

The `deployUncheckedPolicy` method deploys a policy that always grants access. If a policy already exists, it is removed and replaced by this policy.

- endDeployPolicies

```
public void endDeployPolicies(DeployPolicyHandle handle) throws
 ResourceCreationException
```

The `deployExcludedPolicy` method deploys a policy that always denies access. If a policy already exists, it is removed and replaced by this policy.

- startDeployPolicies

```
public deployPolicyHandle startDeployPolicies(ApplicationInfo application)
 throws DeployHandleCreationException
```

The `startDeployPolicies` method marks the beginning of an application policy deployment and is called on all servers within a WebLogic Server domain where an application is targeted.

- undeployAllPolicies

```
public void undeployAllPolicies(DeployPolicyHandle handle) throws
 ResourceRemovalException
```

The `undeployAllPolicies` method deletes a set of policy definitions on behalf of an undeployed Web application or EJB.

See *Java API Reference for Oracle WebLogic Server* for the DeployableAuthorizationProviderV2 SSPI.

The ApplicationInfo Interface

The ApplicationInfo interface passes data about an application deployment to a security provider. You can use this data to uniquely identity the application.

The Security Framework implements the ApplicationInfo interface for your convenience. You do not need to implement any methods for this interface.

The `DeployableAuthorizationProviderV2` and `DeployableRoleProviderV2` interfaces use `ApplicationInfo`. For example, consider an implementation of the `DeployableAuthorizationProviderV2` methods. The Security Framework calls the `DeployableAuthorizationProviderV2` `startDeployPolicies` method and passes in the ApplicationInfo interface for this application. The `ApplicationInfo` data is determined based on the information supplied in the WebLogic Remote Console when an application is deployed.

The `startDeployPolicies` method returns `DeployPolicyHandle`, which you can then use in the other `DeployableAuthorizationProviderV2` methods.

You use the `ApplicationInfo` interface to get the application identifier, the component name, and the component type for this application. Component type can be `APPLICATION`, `CONTROL_RESOURCE`, `EJB`, or `WEBAPP`, as defined in the `ApplicationInfo.ComponentType` class.

The following example shows one way to accomplish this task:

```
public DeployPolicyHandle startDeployPolicies(ApplicationInfo appInfo)
    throws DeployHandleCreationException
     :
// Obtain the application information...
    String appId = appInfo.getApplicationIdentifier();
    ComponentType compType = appInfo.getComponentType();
    String compName = appInfo.getComponentName();
```

The Security Framework calls the `DeployableAuthorizationProviderV2` `deleteApplicationPolicies` method and passes in the `ApplicationInfo` interface for this application. The `deleteApplicationPolicies` method deletes all policies for an application and is called (only on the Administration Server within a WebLogic Server domain) at the time an application is deleted.

Implement the AccessDecision SSPI

When you implement the `AccessDecision` SSPI, you must provide implementations for the following methods:

- isAccessAllowed

  ```
  public Result isAccessAllowed(Subject subject, Map roles,
  Resource resource, ContextHandler handler, Direction direction) throws
  InvalidPrincipalException
  ```

  The `isAccessAllowed` method utilizes information contained within the subject to determine if the requestor should be allowed to access a protected method. The `isAccessAllowed` method may be called prior to or after a request, and returns values of `PERMIT`, `DENY`, or `ABSTAIN`. If multiple Access Decisions are configured and return conflicting values, an adjudication provider will be needed to determine a final result. For more information, see Adjudication Providers.

- isProtectedResource

  ```
  public boolean isProtectedResource(Subject subject, Resource resource) throws
  InvalidPrincipalException
  ```

  The `isProtectedResource` method is used to determine whether the specified WebLogic resource is protected, without incurring the cost of an actual access check. It is only a lightweight mechanism because it does not compute a set of security roles that may be granted to the caller's subject.

See *Java API Reference for Oracle WebLogic Server* for the AccessDecision SSPI.

Example: Creating the Runtime Class for the Sample Authorization Provider

Example 6-1 shows the `SampleAuthorizationProviderImpl.java` class, which is the runtime class for the sample authorization provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription` and `shutdown` (as described in Understand the Purpose of the Provider SSPIs.)
- The method inherited from the `AuthorizationProvider` SSPI: the `getAccessDecision` method (as described in Implement the AuthorizationProvider SSPI).
- The seven methods in the `DeployableAuthorizationProviderV2` SSPI: the `deleteApplicationPolicies`, `deployExcludedPolicy`, `deployPolicy`, `deployUncheckedPolicy`, `endDeployPolicies`, `starteployPolicies`, and `undeployAllPolicies` methods (as described in Implement the DeployableAuthorizationProviderV2 SSPI).
- The two methods in the `AccessDecision` SSPI: the `isAccessAllowed` and `isProtectedResource` methods (as described in Implement the AccessDecision SSPI).

> **Note**
>
> The bold face code in Example 6-1 highlights the class declaration and the method signatures.

Example 6-1 SimpleSampleAuthorizationProviderImpl.java

```
package examples.security.providers.authorization.simple;
import java.security.Principal;
import java.util.Date;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import jakarta.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.SubjectUtils;
import weblogic.security.WLSPrincipals;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AccessDecision;
import weblogic.security.spi.ApplicationInfo;
import weblogic.security.spi.ApplicationInfo.ComponentType;
import weblogic.security.spi.DeployableAuthorizationProviderV2;
import weblogic.security.spi.DeployPolicyHandle;
import weblogic.security.spi.Direction;
import weblogic.security.spi.InvalidPrincipalException;
import weblogic.security.spi.Resource;
import weblogic.security.spi.Result;
import weblogic.security.spi.SecurityServices;
import weblogic.security.spi.VersionableApplicationProvider;
public final class SimpleSampleAuthorizationProviderImpl implements
DeployableAuthorizationProviderV2, AccessDecision, VersionableApplicationProvider
{
   private static String[] NO_ACCESS = new String[0];
   private static String[] ALL_ACCESS = new String[] {WLSPrincipals.getEveryoneGroupname()};
   private String description;
   private SimpleSampleAuthorizerDatabase database;
   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SimpleSampleAuthorizationProviderImpl.initialize");
      SimpleSampleAuthorizerMBean myMBean = (SimpleSampleAuthorizerMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
      database = new SimpleSampleAuthorizerDatabase(myMBean);
   }
   public String getDescription()
   {
      return description;
   }
   public void shutdown()
   {
      System.out.println("SampleAuthorizationProviderImpl.shutdown");
   }
   public AccessDecision getAccessDecision()
   {
      return this;
   }
   public Result isAccessAllowed(Subject subject, Map roles, Resource resource,
   ContextHandler handler, Direction direction)
   {
      System.out.println("SimpleSampleAuthorizationProviderImpl.isAccessAllowed");
```

```
      System.out.println("\tsubject\t= " + subject);
      System.out.println("\troles\t= " + roles);
      System.out.println("\tresource\t= " + resource);
      System.out.println("\tdirection\t= " + direction);
      Set principals = subject.getPrincipals();
      for (Resource res = resource; res != null; res = res.getParentResource()) {
         if (database.policyExists(res)) {
            Result result = isAccessAllowed(res, subject, roles);
            System.out.println("\tallowed\t= " + result);
            return result;
         }
      }
      Result result = Result.ABSTAIN;
      System.out.println("\tallowed\t= " + result);
      return result;
   }
   public boolean isProtectedResource(Subject subject, Resource resource) throws
   InvalidPrincipalException
   {
      System.out.println("SimpleSampleAuthorizationProviderImpl.
        isProtectedResource");
      System.out.println("\tsubject\t= " + subject);
      System.out.println("\tresource\t= " + resource);
      for (Resource res = resource; res != null; res = res.getParentResource()) {
         if (database.policyExists(res)) {
            System.out.println("\tprotected\t= true");
            return true;
         }
      }
      System.out.println("\tprotected\t= false");
      return false;
   }
public DeployPolicyHandle startDeployPolicies(ApplicationInfo application)
{
   String appId = application.getApplicationIdentifier();
   String compName = application.getComponentName();
   ComponentType compType = application.getComponentType();
   DeployPolicyHandle handle = new    SampleDeployPolicyHandle(appId,compName,compType);
   database.removePoliciesForComponent(appId, compName, compType);
   return handle;
  public void deployPolicy(DeployPolicyHandle handle,
Resource resource, String[] roleNamesAllowed)
{
   System.out.println("SimpleSampleAuthorizationProviderImpl.deployPolicy");
   System.out.println("\thandle\t= " + ((SampleDeployPolicyHandle)handle).toString());
   System.out.println("\tresource\t= " + resource);
   for (int i = 0; roleNamesAllowed != null && i < roleNamesAllowed.length; i++) {
     System.out.println("\troleNamesAllowed[" + i + "]\t= " + roleNamesAllowed[i]);
}
database.setPolicy(resource, roleNamesAllowed);
}
   public void deployUncheckedPolicy(DeployPolicyHandle handle, Resource resource)
{
   System.out.println("SimpleSampleAuthorizationProviderImpl.deployUncheckedPolicy");
   System.out.println("\thandle\t= " + ((SampleDeployPolicyHandle)handle).toString());
   System.out.println("\tresource\t= " + resource);
   database.setPolicy(resource, ALL_ACCESS);
}
public void deployExcludedPolicy(DeployPolicyHandle handle, Resource resource)
  {
   System.out.println("SimpleSampleAuthorizationProviderImpl.deployExcludedPolicy");
   System.out.println("\thandle\t= " + ((SampleDeployPolicyHandle)handle).toString());
```

```
   System.out.println("\tresource\t= " + resource);
   database.setPolicy(resource, NO_ACCESS);
}
public void endDeployPolicies(DeployPolicyHandle handle)
{
   database.savePolicies();
}
public void undeployAllPolicies(DeployPolicyHandle handle)
{
   System.out.println("SimpleSampleAuthorizationProviderImpl.undeployAllPolicies");
   SampleDeployPolicyHandle myHandle = (SampleDeployPolicyHandle)handle;
   System.out.println("\thandle\t= " + myHandle.toString());

   // remove policies
   database.removePoliciesForComponent(myHandle.getApplication(),
                                       myHandle.getComponent(),
                                       myHandle.getComponentType());
}
public void deleteApplicationPolicies(ApplicationInfo application)
{
   System.out.println("SimpleSampleAuthorizationProviderImpl.deleteApplicationPolicies");
   String appId = application.getApplicationIdentifier();
   System.out.println("\tapplication identifier\t= " + appId);

   // clear out policies for the application
   database.removePoliciesForApplication(appId);
}
private boolean rolesOrSubjectContains(Map roles, Subject subject, String roleOrPrincipalWant)
{
   // first, see if it's a role name match
if (roles.containsKey(roleOrPrincipalWant)) {
     return true;
}

   // second, see if it's a group name match
   if (SubjectUtils.isUserInGroup(subject, roleOrPrincipalWant)) {
     return true;
   }

   // third, see if it's a user name match
   if (roleOrPrincipalWant.equals(SubjectUtils.getUsername(subject))) {
     return true;
   }

   // didn't match
   return false;
}

private Result isAccessAllowed(Resource resource, Subject subject, Map roles)
{
   // loop over the principals and roles in our database who are allowed to access this resource
   for (Enumeration e = database.getPolicy(resource); e.hasMoreElements();) {
     String roleOrPrincipalAllowed = (String)e.nextElement();
     if (rolesOrSubjectContains(roles, subject, roleOrPrincipalAllowed)) {
       return Result.PERMIT;
     }
   }
   // the resource was explicitly mentioned and didn't grant access
   return Result.DENY;
}

public void createApplicationVersion(String appId, String sourceAppId)
```

```
{
   System.out.println("SimpleSampleAuthorizationProviderImpl.createApplicationVersion");
   System.out.println("\tapplication identifier\t= " + appId);
   System.out.println("\tsource app identifier\t= " + ((sourceAppId != null) ? sourceAppId : "None"));

   // create new policies when existing application is specified
   if (sourceAppId != null) {
     database.clonePoliciesForApplication(sourceAppId,appId);
   }
}
public void deleteApplicationVersion(String appId)
{
   System.out.println("SimpleSampleAuthorizationProviderImpl.deleteApplicationVersion");
   System.out.println("\tapplication identifier\t= " + appId);

   // clear out policies for the application
   database.removePoliciesForApplication(appId);
}

public void deleteApplication(String appName)
{
   System.out.println("SimpleSampleAuthorizationProviderImpl.deleteApplication");
   System.out.println("\tapplication name\t= " + appName);

   // clear out policies for the application
   database.removePoliciesForApplication(appName);
}

class SampleDeployPolicyHandle implements DeployPolicyHandle
{
   Date date;
   String application;
   String component;
   ComponentType componentType;

   SampleDeployPolicyHandle(String app, String comp, ComponentType type)
{
     this.application = app;
     this.component = comp;
     this.componentType = type;
     this.date = new Date();
}

     public String getApplication() { return application; }
     public String getComponent() { return component; }
     public ComponentType getComponentType() { return componentType; }

     public String toString()
     {
       String name = component;
       if (componentType == ComponentType.APPLICATION)
          name = application;
       return componentType +" "+ name +" ["+ date.toString() +"]";
     }
   }
}
```

Policy Consumer SSPI

WebLogic Server implements a policy consumer for JMX (MBean) default policies and Web service annotations. This release of WebLogic Server includes an SSPI that authorization providers can use to obtain the policy collections.

The `PolicyConsumer` SSPI is optional; only those authorization providers that implement the SSPI are called to consume a policy collection.

The SSPI supports both the delivery of initial policy collections and the delivery of updated policy collections.

All authorization providers that support the `PolicyConsumer` SSPI are called to consume a policy collection. Each authorization provider can choose to skip or obtain the policy collection for a given policy set. In the case where a provider persists policy, the provider need only collect the policy once. However, providers keeping policy in memory can obtain the policy collection again.

The out-of-the-box WebLogic Server Authorization providers persist the policy into LDAP.

Required SSPI Interfaces

If you want your custom authorization provider to support the delivery of policy collections, you must implement three interfaces:

- `weblogic.security.spi.PolicyConsumerFactory`
- `weblogic.security.spi.PolicyConsumer`
- `weblogic.security.spi.PolicyCollectionHandler`
- These interfaces are described in the sections that follow.

Implement the PolicyConsumerFactory SSPI Interface

An authorization provider implements the `PolicyConsumerFactory` interface so that an instance of a PolicyConsumer is available to the WebLogic Security Framework. The WebLogic Security Framework calls your `PolicyConsumerFactory` implementation to obtain the provider's implementation of the policy consumer.

The `PolicyConsumerFactory` SSPI has one method, which returns your implementation of the `PolicyConsumer` SSPI interface.

```
public interface PolicyConsumerFactory
{
/**
* Obtain the implementation of the PolicyConsumer
* security service provider interface (SSPI).
*
* @return a PolicyConsumer SSPI implementation.
*/
public PolicyConsumer getPolicyConsumer();
}
```

Implement the PolicyConsumer SSPI Interface

The `PolicyConsumer` SSPI returns a policy collection handler for consumption of a policy collection. It has one method, `getPolicyCollectionHandler()`, which takes a

PolicyCollectionInfo implementation as an argument and returns your implementation of the PolicyCollectionHandler interface.

```
public interface PolicyConsumer
  {
    /**
     * Obtain a policy handler for consumption of a policy set.
     *
     * @param info the PolicyCollectionInfo for the policy set.
     *
     * @return a PolicyCollectionHandler or NULL which indicates
     *     that the policy set is not needed.
     *
     * @exception ConsumptionException if an error occurs
     *     obtaining the handler and the policy set cannot be consumed.
     */
public PolicyCollectionHandler getPolicyCollectionHandler(
           PolicyCollectionInfo info)
    throws ConsumptionException;
}
```

The WebLogic Security Framework calls the getPolicyCollectionHandler() method and passes data about a policy collection to a security provider as an implementation of the PolicyCollectionInfo interface. (This interface implementation is provided for you, you do not have to implement it.)

You use the PolicyCollectionInfo getName(), getVersion(), getTimestamp(), and getResourceTypes() methods to discover information about this policy set. You then return a PolicyCollectionHandler, or NULL to indicate that the policy collection is not needed.

```
public interface PolicyCollectionInfo
{
/**
 * Get the name of the collection.
 */
public String getName();

/**
 * Get the runtime version of the policy.
 */
public String getVersion();

/**
 * Get the timestamp of the policy.
 */
public String getTimestamp();

/**
 * Get the resource types used in the policy collection.
 */
public Resource[] getResouceTypes();
}
```

Implement the PolicyCollectionHandler SSPI Interface

The PolicyConsumer.getPolicyCollectionHandler() method returns your implementation of the PolicyCollectionHandler interface. PolicyCollectionHandler has three methods: setPolicy, setUncheckedPolicy, and done(). The setPolicy() method takes a resource and role names and sets a policy based on the role. The setUncheckedPolicy() method opens access to everyone.

The `done()` method signals the completion of the policy collection. We recommend that the `done()` method remove all old policies for the policy set.

```
public interface PolicyCollectionHandler
{
     /**
      * Set a policy for the specified resource.
      */
     public void setPolicy(Resource resource, String[] roleNames)
        throws ConsumptionException;

     /**
      * Sets a policy which always grants access.
      */
     public void setUncheckedPolicy(Resource resource)
        throws ConsumptionException;

     /**
      * Signals the completion of the policy collection.
      */
     public void done()
       throws ConsumptionException;

}
```

Supporting an Updated Policy Collection

To support the delivery of an updated policy collection, all authorization providers that support the `PolicyConsumer` SSPI need to examine the contents of the `PolicyCollectionInfo` passed in the `PolicyConsumer.getPolicyCollectionHandler()` method to determine if a policy set has changed. Each provider must decide (possibly by configuration) how to perform conflict resolution with the initial policy collection and any customized policy received outside of the SSPI.

For the WebLogic Server supplied authorization providers, customized policy will not be replaced by the updated policy collection: all policy from the initial policy collection will be removed and only the customized policies, plus the updated policy collection, will be in effect. If the policy collection info has a different timestamp or version, it's treated as an updated policy collection. The collection name is used as a persistence key.

The PolicyConsumerMBean

Authorization providers that implement the Policy Consumer SSPI must also implement the `weblogic.management.security.authorization.PolicyConsumerMBean` to indicate that the provider supports policy consumption.

PolicyStoreMBean

This release of WebLogic Server includes support for a new MBean (`weblogic.management.security.authorization.PolicyStoreMBean`) that allows for standard management (add, delete, get, list, modify, read) of administrator-generated XACML policies and policy sets. An authorization or role mapping provider MBean can optionally implement this MBean interface.

The PolicyStoreMBean methods allow security administrators to manage policy in the server as XACML documents. This includes creating and managing a domain that uses the default XACML provider, as well as managing XACML documents that the administrator has created. The administrator can then use WLST to manage these XACML policies in WebLogic Server.

WebLogic Server includes an implementation of this MBean for use with the out-of-the-box XACML providers, and you can write your own implementation of this MBean for use with your own custom authorization or role mapping providers. The WebLogic Server out-of-the-box XACML providers support the mandatory features of XACML, as described in the XACML 2.0 Core Specification (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf), with the Oracle-specific usage described in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Policies are expressed as XACML 2.0 Policy or PolicySet documents. Custom authorization providers should expect standard Policy or PolicySet documents as described in the XACML 2.0 Core Specification (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf). Custom role mapping providers should expect Policy or PolicySet documents consistent with role assignment policies described by the Core and hierarchical role based access control (RBAC) profile of XACML v2.0 (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf).

Specifically, the Target must contain:

- An `ActionAttributeDesignator` with the id, `urn:oasis:names:tc:xacml:1.0:action:action-id`, and the value, `urn:oasis:names:tc:xacml:2.0:actions:enableRole`, according to `anyURI-equal`. For example:

```
<Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal">

<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">urn:oasis:names:tc:xacml:2.0
:actions:enableRole
</AttributeValue>

<ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="true"/>

</ActionMatch>
</Action>
```

- A `ResourceAttributeDesignator` with the id, `urn:oasis:names:tc:xacml:2.0:subject:role`, and a value naming the role being assigned, according to string-equal. For example:

```
<ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0:resource:resource-
ancestor-or-self"
DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"/>
```

Examining the Format of a XACML Policy File

The XACML 2.0 Core Specification (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf) and the Oracle extensions described in *Securing Resources Using Roles and Policies for Oracle WebLogic Server* are the definitive sources of information for the XACML policy files used by the supplied XACML Authorization and Role Mapping Providers.

However, as part of your development process, you may want to take a look at the format of a supported XACML file. Use WLST to export the data from the XACML Authorization or Role Mapping provider's database as a XACML file. Copy this exported XACML file to a file with some other name and use the tool of your choice to review the copy.

> **Note**
>
> Treat the exported file as read-only. If you do make changes, do not import the file back into WebLogic Server. Editing exported files might result in an unusable WebLogic Server configuration and is not supported.

Using WLST to Add a Policy to the PolicyStoreMBean

Example 6-2 shows an example of using WLST to add a single policy to an instance of the PolicyStoreMBean from a XACML file.

The example assumes that you have defined the properties used in this script elsewhere, in a manner similar to the following lines from an `ant` script:

```
<property name="xacml-docs-dir" value="${xacmldir}/xacml-docs"/>
<sysproperty key="file" value="${xacml-docs-dir}/policy-getSubject.xacml"/>
```

You should avoid entering clear-text passwords in WLST commands in general, and you should especially avoid saving on disk WLST scripts that include clear-text passwords. In these instances you should use a mechanism for passing encrypted passwords instead. See Security for WLST in *Understanding the WebLogic Scripting Tool*.

Example 6-2 Using WLST to Add a Policy to the PolicyStoreMBean

```
:
try:
      protocol = System.getProperty("protocol")
      host = System.getProperty("host")
      user = System.getProperty("authuser")
      passwd = System.getProperty("authpwd")
      port = System.getProperty("port")
      dom = System.getProperty("domain")
      rlm = System.getProperty("realm")
      fil = System.getProperty("file")
      prov = System.getProperty("provider")
      stat = System.getProperty("status")

def configure():
try:
      url = protocol + "://" + host + ":" + port
      connect(user,passwd, url)
      path = "/SecurityConfiguration/" + dom + "/Realms/" + rlm + "/" + prov
      print("cd'ing to " + path)
      cd(path)
      print("calling open()")
      xacmlFile = open(fil,"r")
      print("calling read()")
      xacmlDoc = xacmlFile.read()
      print("calling cmo.addPolicy")
      if stat == "none":
          cmo.addPolicy(xacmlDoc)
      else:
          cmo.addPolicy(xacmlDoc, stat)
      print("Add error handling")
:
:
```

As described in the Navigating and Interrogating MBeans section of *Understanding the WebLogic Scripting Tool*, when WLST first connects to an instance of WebLogic Server, the variable, `cmo` (Current Management Object), is initialized to the root of all configuration management objects, `DomainMBean`. When you navigate to an MBean type, in this case `SecurityConfigurationMBean`, the value of `cmo` reflects `SecurityConfigurationMBean`. When you navigate to an MBean instance, in this case to an Authorizer MBean that implements the PolicyStoreMBean, identified in the example by the variable `prov`, WLST changes the value of `cmo` to be the current MBean instance.

The example uses the addPolicy() method of the PolicyStoreMBean to add a policy read from a XACML file to the policy store. Two variants of the addPolicy() method (without and with status) are shown.

If you use an addPolicy() method that does not specify status, it defaults to ACTIVE, which indicates that the policy is evaluated for any decision to which its target applies. You can explicitly set status to be ACTIVE, INACTIVE, or BYREFERENCE. The INACTIVE status indicates that the policy will never be evaluated and is only being stored. The BYREFERENCE status indicates that the policy will only be evaluated when referenced by a policy set that is being evaluated.

You can invoke this type of WLST script from the command line, in a manner similar to the following:

```
java -Dhost="localhost " -Dprotocol="t3" -Dauthuser="weblogic"
-Dauthpwd="weblogic" -Dport="7001" -Ddomain="mydomain" -Drealm="myrealm"
-Dprovider="Authorizers/XACMLAuthorizer"
-Dfile="C:/XACML/xacml-docs/policy12.xml" -Dstatus="none" weblogic.WLST
XACML/scripts/XACMLaddPolicy.py
```

Using WLST to Read a PolicySet as a String

Example 6-3 shows an example of using WLST to read a PolicySet as a string.

The example assumes that you have defined the properties used in this script elsewhere, in a manner similar to the following lines from an `ant` script:

```
<sysproperty key="identifier"
value="urn:sample:xacml:2.0:wlssecqa:resource:type@E@Fejb@G@M@Oapplication@ENoD
DRolesOrPoliciesEar@M@Omodule@Eejb11inEarMiniAppBean.jar@M@Oejb@EMiniAppBean@
M@Omethod@EgetSubject@M@OmethodInterface@ERemote"/>
<sysproperty key="version" value="1.0"/>
```

You should avoid entering clear-text passwords in WLST commands in general, and you should especially avoid saving on disk WLST scripts that include clear-text passwords. In these instances you should use a mechanism for passing encrypted passwords instead. See Security for WLST in *Understanding the WebLogic Scripting Tool*.

Example 6-3 Using WLST to Read a PolicySet as a String

```
:
:
try:
      print("start XACMLreadPolicySet.py")
      protocol = System.getProperty("protocol")
      host = System.getProperty("host")
      user = System.getProperty("authuser")
      passwd = System.getProperty("authpwd")
      port = System.getProperty("port")
      dom = System.getProperty("domain")
      rlm = System.getProperty("realm")
      prov = System.getProperty("provider")
```

```
      id = System.getProperty("identifier")
      vers = System.getProperty("version")
:
:
def configure():
try:
      url = protocol + "://" + host + ":" + port
      connect(user,passwd, url)
      path = "/SecurityConfiguration/" + dom + "/Realms/" + rlm + "/" + prov
      print("cd'ing to " + path)
      cd(path)
      polset = cmo.readPolicySetAsString(id, vers)
      print("readPolicySetAsString() returned the following policy set: " + polset)
      print"Add error handling."
:
:
```

As described in the XACML 2.0 Core Specification (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf), the <PolicySet> element contains a set of <Policy> or other <PolicySet> elements and a specified procedure for combining the results of their evaluation. See the XACML 2.0 Core Specification for complete information.

Bulk Authorization Providers

This release of WebLogic Server includes bulk access versions of the following authorization provider SSPI interfaces:

- `BulkAuthorizationProvider`
- `BulkAccessDecision`

The bulk access SSPI interfaces allow authorization providers to receive multiple decision requests in one call rather than through multiple calls, typically in a `'for'` loop. The intent of the bulk SSPI variants is to allow provider implementations to take advantage of internal performance optimizations, such as detecting that many of the passed-in `Resource` objects are protected by the same policy and will generate the same decision result.

There are subtle differences in how the non-bulk and bulk versions of the SSPI interfaces are used.

Note that the `BulkAccessDecision.isAccessAllowed()` method takes a `Map` of roles, indexed first by `Resource` object and then by role name (`Map<Resource, Map<String, SecurityRole>> roles`), that are associated with the subject and should be taken into consideration when making the authorization decision.

The `BulkAccessDecision.isAccessAllowed()` method returns a `Map (indexed by Resource, result)` that indicates whether the authorization policies defined for the resources allow the requested methods to be performed.

Configure a Custom Authorization Provider

Configuring a custom authorization provider means that you are adding the custom authorization provider to your security realm, where it can be accessed by applications requiring authorization services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers. This section contains information that is important for the person configuring your custom authorization providers:

- Managing Authorization Providers and Deployment Descriptors
- Enabling Security Policy Deployment

> **Note**
>
> The steps for configuring a custom authorization provider are described in Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

Managing Authorization Providers and Deployment Descriptors

Some application components, such as Enterprise JavaBeans (EJBs) and Web applications, store relevant deployment information in Java EE and WebLogic Server deployment descriptors. For Web applications, the deployment descriptor files (called `web.xml` and `weblogic.xml`) contain information for implementing the Java EE security model, including declarations of security policies. Typically, you will want to include this information when first configuring your authorization providers in the WebLogic Remote Console.

Because the Java EE platform standardizes Web application and EJB security in deployment descriptors, WebLogic Server integrates this standard mechanism with its Security Service to give you a choice of techniques for securing Web application and EJB resources. You can use deployment descriptors exclusively, the WebLogic Remote Console exclusively, or you can combine the techniques for certain situations.

Depending on the technique you choose, you also need to apply a Security Model. WebLogic supports different security models for individual deployments, and a security model for realm-wide configurations that incorporate the technique you want to use.

When configured to use deployment descriptors, WebLogic Server reads security policy information from the `web.xml` and `weblogic.xml` deployment descriptor files (examples of `web.xml` and `weblogic.xml` files are shown in Example 6-4 and Example 6-5). This information is then copied into the security provider database for the authorization provider.

Example 6-4 Sample web.xml File

```
<web-app>
   <welcome-file-list>
      <welcome-file>welcome.jsp</welcome-file>
   </welcome-file-list>
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Success</web-resource-name>
         <url-pattern>/welcome.jsp</url-pattern>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
         <role-name>developers</role-name>
      </auth-constraint>
   </security-constraint>
   <login-config>
       <auth-method>BASIC</auth-method>
       <realm-name>default</realm-name>
   </login-config>
   <security-role>
      <role-name>developers</role-name>
```

```
   </security-role>
</web-app>
```

Example 6-5 Sample weblogic.xml File

```
<weblogic-web-app>
   <security-role-assignment>
      <role-name>developers</role-name>
      <principal-name>myGroup</principal-name>
   </security-role-assignment>
</weblogic-web-app>
```

Enabling Security Policy Deployment

If you implemented the `DeployableAuthorizationProviderV2` SSPI as part of developing your custom authorization provider and want to support deployable security policies, the person configuring the custom authorization provider (that is, you or an administrator) must be sure that the Policy Deployment Enabled check box in the WebLogic Remote Console is checked. Otherwise, deployment for the authorization provider is considered turned off. Therefore, if multiple authorization providers are configured, the Policy Deployment Enabled check box can be used to control which authorization provider is used for security policy deployment.

Provide a Mechanism for Security Policy Management

While configuring a custom authorization provider using the WebLogic Remote Console makes it accessible by applications requiring authorization services, you also need to supply administrators with a way to manage this security provider's associated security policies. The WebLogic Authorization provider, for example, supplies administrators with a Policy Editor page that allows them to add, modify, or remove security policies for various WebLogic resources.

Neither the Policy Editor page nor access to it is available to administrators when you develop a custom authorization provider. Therefore, you must provide your own mechanism for security policy management. This mechanism must read and write security policy data (that is, expressions) to and from the custom authorization provider's database.

For more information, see Develop a Stand-Alone Tool for Security Policy Management.

Develop a Stand-Alone Tool for Security Policy Management

You would typically select this option if you want to develop a tool that is entirely separate from the WebLogic Remote Console.

For this option, you do not need to develop any management MBeans. However, your tool needs to:

1. Determine the WebLogic resource's ID. See WebLogic Resource Identifiers.
2. Determine how to represent the expressions that make up a security policy. (This representation is entirely up to you and need not be a string.)
3. Read and write the expressions from and to the custom authorization provider's database.

7

Adjudication Providers

This chapter describes adjudication provider concepts and functionality, and provides step-by-step instructions for developing a custom adjudication provider.
Adjudication involves resolving any authorization conflicts that may occur when more than one authorization provider is configured, by weighing the result of each authorization provider's Access Decision. In WebLogic Server, an adjudication provider is used to tally the results that multiple Access Decisions return, and determines the final `PERMIT` or `DENY` decision. An adjudication provider may also specify what should be done when an answer of `ABSTAIN` is returned from a single authorization provider's Access Decision.

This chapter includes the following sections:

The Adjudication Process

The use of adjudication providers is part of the authorization process, and is described in The Authorization Process.

Do You Need to Develop a Custom Adjudication Provider?

The default (that is, active) security realm for WebLogic Server includes a WebLogic Adjudication provider. The WebLogic Adjudication provider is responsible for adjudicating between potentially differing results rendered by multiple authorization providers' Access Decisions, and rendering a final verdict on whether or not access will be granted to a WebLogic resource.

The WebLogic Adjudication provider has an attribute called Require Unanimous Permit that governs its behavior. By default, the Require Unanimous Permit attribute is set to `TRUE`, which causes the WebLogic Adjudication provider to act as follows:

- If all the authorization providers' Access Decisions return `PERMIT`, then return a final verdict of `TRUE` (that is, permit access to the WebLogic resource).
- If some authorization providers' Access Decisions return `PERMIT` and others return `ABSTAIN`, then return a final verdict of `FALSE` (that is, deny access to the WebLogic resource).
- If any of the authorization providers' Access Decisions return `ABSTAIN` or `DENY`, then return a final verdict of `FALSE` (that is, deny access to the WebLogic resource).

If you change the Require Unanimous Permit attribute to `FALSE`, the WebLogic Adjudication provider acts as follows:

- If all the authorization providers' Access Decisions return `PERMIT`, then return a final verdict of `TRUE` (that is, permit access to the WebLogic resource).

- If some authorization providers' Access Decisions return `PERMIT` and others return `ABSTAIN`, then return a final verdict of `TRUE` (that is, permit access to the WebLogic resource).
- If any of the authorization providers' Access Decisions return `DENY`, then return a final verdict of `FALSE` (that is, deny access to the WebLogic resource).

> **Note**
>
> You set the Require Unanimous Permit attributes when you configure the WebLogic Adjudication provider. See Configuring the WebLogic Adjudication Provider in *Administering Security for Oracle WebLogic Server*.

If you want an adjudication provider that behaves in a way that is different from what is described above, then you need to develop a custom adjudication provider. (Keep in mind that an adjudication provider may also specify what should be done when an answer of `ABSTAIN` is returned from a single authorization provider's Access Decision, based on your specific security requirements.)

How to Develop a Custom Adjudication Provider

If the WebLogic Adjudication provider does not meet your needs, you can develop a custom adjudication provider by following these steps:

1. Create Runtime Classes Using the Appropriate SSPIs, or, optionally, use the Bulk Adjudication Providers
2. Generate an MBean type for your custom adjudication provider by completing the steps described in Generate an MBean Type Using the WebLogic MBeanMaker.
3. Configure the Custom Adjudication Provider

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom adjudication provider by following these steps:

- Implement the AdjudicationProviderV2 SSPI
- Implement the AdjudicatorV2 SSPI

Implement the AdjudicationProviderV2 SSPI

To implement the `AdjudicationProviderV2` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following method:

- getAdjudicator

```
public AdjudicatorV2 getAdjudicator()
```

The `getAdjudicator` method obtains the implementation of the `AdjudicatorV2` SSPI. For a single runtime class called `MyAdjudicationProviderImpl.java`, the implementation of the `getAdjudicator` method would be:

```
return this;
```

If there are two runtime classes, then the implementation of the `getAdjudicator` method could be:

```
return new MyAdjudicatorImpl;
```

This is because the runtime class that implements the `AdjudicationProviderV2` SSPI is used as a factory to obtain classes that implement the `AdjudicatorV2` SSPI.

See *Java API Reference for Oracle WebLogic Server* for the AdjudicationProviderV2 SSPI.

Implement the AdjudicatorV2 SSPI

To implement the `AdjudicatorV2` SSPI, provide implementations for the following methods:

- initialize

  ```
  public void initialize(AuthorizerMBean[] accessDecisionClassNames)
  ```

  The `initialize` method initializes the names of all the configured authorization providers' Access Decisions that will be called to supply a result for the "is access allowed?" question. The `accessDecisionClassNames` parameter may also be used by an adjudication provider in its `adjudicate` method to favor a result from a particular Access Decision. For more information about authorization providers and Access Decisions, see Authorization Providers.

- adjudicate

  ```
  public boolean adjudicate(Result[] results, Resource resource,
                            ContextHandler handler)
  ```

  The `adjudicate` method determines the answer to the "is access allowed?" question, given all the results from the configured authorization providers' Access Decisions.

See *Java API Reference for Oracle WebLogic Server* for the AdjudicatorV2 SSPI.

Bulk Adjudication Providers

This release of WebLogic Server includes bulk access versions of the following adjudication provider SSPI interfaces:

- `BulkAdjudicationProvider`
- `BulkAdjudicator`

The bulk access SSPI interfaces allow adjudication providers to receive multiple decision requests in one call rather than through multiple calls, typically in a `'for'` loop. The intent of the bulk SSPI variants is to allow provider implementations to take advantage of internal performance optimizations, such as detecting that many of the passed-in `Resource` objects are protected by the same policy and will generate the same decision result.

There are subtle differences in how the non-bulk and bulk versions of the SSPI interfaces are used.

The `BulkAdjudicator.adjudicate()` method takes a `List` of Map `(Resource, Result)` instances, as passed in by the WebLogic Server Authorization Manager, which contain the

results of each bulk access decision. The order of results is the same as the order of the Access Decision class names that were passed in the `BulkAdjudicator.initialize()` method.

Note too that the `BulkAdjudicator.adjudicate()` method returns a `Set` of `Resource` objects. If a `Resource` object is present in the set, access has been granted for that object; otherwise, access has been denied.

Configure the Custom Adjudication Provider

Configuring a custom adjudication provider means that you are adding the custom adjudication provider to your security realm, where it can be accessed by applications requiring adjudication services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers. The steps for configuring a custom adjudication provider are described in Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

8

Role Mapping Providers

This chapter describes role mapping provider concepts and functionality, and provides step-by-step instructions for developing a custom role mapping provider.
Role mapping is the process whereby principals (users or groups) are dynamically mapped to security roles at runtime. In WebLogic Server, a role mapping provider determines what security roles apply to the principals stored a subject when the subject is attempting to perform an operation on a WebLogic resource. Because this operation usually involves gaining access to the WebLogic resource, role mapping providers are typically used with authorization providers.

This chapter includes the following sections:

Role Mapping Concepts

Before you develop a role mapping provider, you need to understand the following concepts:

Security Roles

A **security role** is a named collection of users or groups that have similar permissions to access WebLogic resources. Like groups, security roles allow you to control access to WebLogic resources for several users at once. However, security roles are scoped to specific resources in a WebLogic Server domain (unlike groups, which are scoped to an entire WebLogic Server domain), and can be defined dynamically (as described in Dynamic Security Role Computation).

> **Note**
>
> See Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*, Security Providers and WebLogic Resources, and Resource Types You Can Secure with Policies in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

The `SecurityRole` interface in the `weblogic.security.service` package is used to represent the abstract notion of a security role. (See *Java API Reference for Oracle WebLogic Server* for the SecurityRole interface.)

Mapping a principal to a security role grants the defined access permissions to that principal, as long as the principal is in the security role. For example, an application may define a security role called `AppAdmin`, which provides write access to a small subset of that application's resources. Any principal in the `AppAdmin` security role would then have write access to those resources. See Dynamic Security Role Computation and Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Many principals can be mapped to a single security role. See Users and Groups, Principals and Subjects.

Security roles are specified in Java EE deployment descriptor files and/or in the WebLogic Remote Console. See Managing Role Mapping Providers and Deployment Descriptors.

Dynamic Security Role Computation

Security roles can be declarative (that is, Java Platform, Enterprise Edition (Java EE) roles) or dynamically computed based on the context of the request.

Dynamic security role computation is the term for this late binding of principals (that is, users or groups) to security roles at runtime. The late binding occurs just prior to an authorization decision for a protected WebLogic resource, regardless of whether the principal-to-security role association is statically defined or dynamically computed. Because of its placement in the invocation sequence, the result of any principal-to-security role computations can be taken as an authentication identity, as part of the authorization decision made for the request.

This dynamic computation of security roles provides a very important benefit: users or groups can be granted a security role based on business rules. For example, a user may be allowed to be in a `Manager` security role only while the actual manager is away on an extended business trip. Dynamically computing this security role means that you do not need to change or redeploy your application to allow for such a temporarily arrangement. Further, you would not need to remember to revoke the special privileges when the actual manager returns, as you would if you temporarily added the user to a `Managers` group.

> **Note**
>
> You typically grant users or groups security roles using the role conditions available in the WebLogic Remote Console. (In this release of WebLogic Server, you cannot write custom role conditions.) See Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

The computed security role is able to access a number of pieces of information that make up the context of the request, including the identity of the target (if available) and the parameter values of the request. The context information is typically used as values of parameters in an expression that is evaluated by the WebLogic Security Framework. This functionality is also responsible for computing security roles that were statically defined through a deployment descriptor or through the WebLogic Remote Console.

> **Note**
>
> The computation of security roles for an authenticated user enhances the Role-Based Access Control (RBAC) security defined by the Java EE specification.
>
> You create dynamic security role computations by defining role statements in the WebLogic Remote Console. See Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

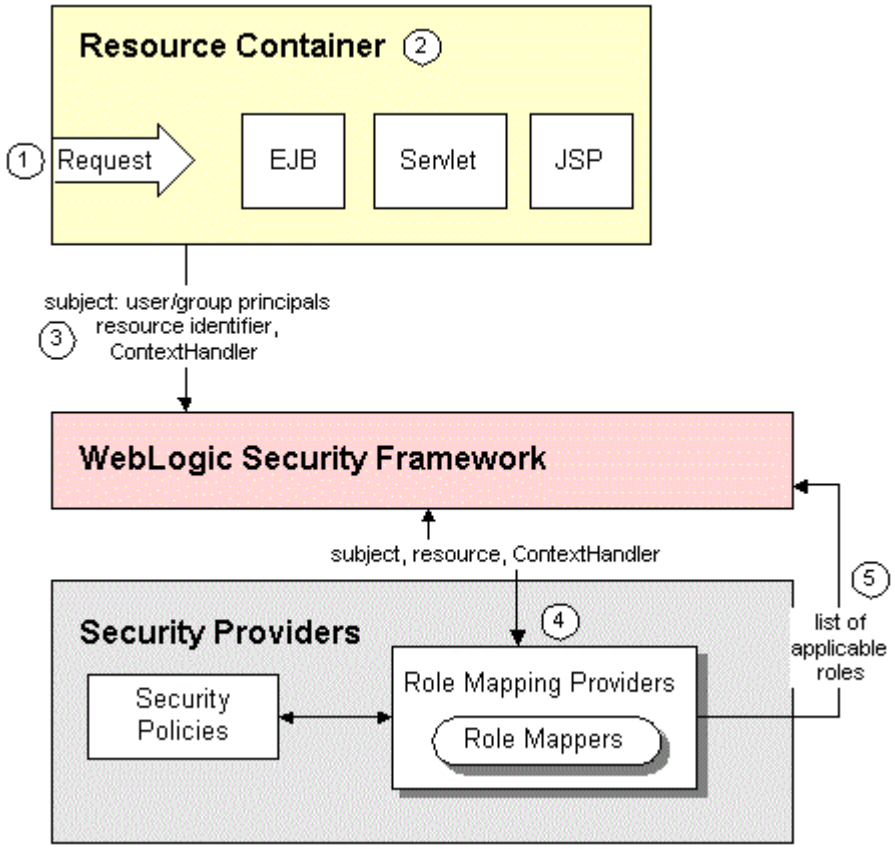
The Role Mapping Process

The WebLogic Security Framework calls each role mapping provider that is configured for a security realm as part of an authorization decision. For related information, see The Authorization Process.

The result of the dynamic security role computation (performed by the role mapping providers) is a set of security roles that apply to the principals stored in a subject at a given moment. These security roles can then be used to make authorization decisions for protected WebLogic resources, as well as for resource container and application code. For example, an Enterprise JavaBean (EJB) could use the Java EE `isCallerInRole` method to retrieve fields from a record in a database, without having knowledge of the business policies that determine whether access is allowed.

Figure 8-1 shows how the role mapping providers interact with the WebLogic Security Framework to create dynamic security role computations, and an explanation follows.

Figure 8-1 Role Mapping Providers and the Role Mapping Process



Generally, role mapping is performed in the following manner:

1. A user or system process requests a WebLogic resource on which it will attempt to perform a given operation.
2. The resource container that handles the type of WebLogic resource being requested receives the request (for example, the EJB container receives the request for an EJB resource).

> **Note**
>
> The resource container could be the container that handles any one of the WebLogic Resources described in Security Providers and WebLogic Resources.

3. The resource container constructs a `ContextHandler` object that may be used by role mapping providers to obtain information associated with the context of the request.

> **Note**
>
> See ContextHandlers and WebLogic Resources.

The resource container calls the WebLogic Security Framework, passing in the subject (which already contains user and group principals), an identifier for the WebLogic resource, and optionally, the `ContextHandler` object (to provide additional input).

> **Note**
>
> See Users and Groups, Principals and Subjects. See WebLogic Resource Identifiers.

4. The WebLogic Security Framework calls each configured role mapping provider to obtain a list of the security roles that apply. This works as follows:
   a. The role mapping providers use the `ContextHandler` to request various pieces of information about the request. They construct a set of `Callback` objects that represent the type of information being requested. This set of `Callback` objects is then passed as an array to the `ContextHandler` using the `handle` method.

      The role mapping providers may call the `ContextHandler` more than once in order to obtain the necessary context information. (The number of times a role mapping provider calls the `ContextHandler` is dependent upon its implementation.)
   b. Using the context information and their associated security provider databases containing security policies, the subject, and the WebLogic resource, the role mapping providers determine whether the requestor (represented by the user and group principals in the subject) is entitled to a certain security role.

      The security policies are represented as a set of expressions or rules that are evaluated to determine if a given security role is to be granted. These rules may require the role mapping provider to substitute the value of context information obtained as parameters into the expression. In addition, the rules may also require the identity of a user or group principal as the value of an expression parameter.

      > **Note**
      >
      > The rules for security policies are set up in the WebLogic Remote Console and in Java EE deployment descriptors. See Security Policies in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

   c. If a security policy specifies that the requestor is entitled to a particular security role, the security role is added to the list of security roles that are applicable to the subject.
   d. This process continues until all security policies that apply to the WebLogic resource or the resource container have been evaluated.
5. The list of security roles is returned to the WebLogic Security Framework, where it can be used as part of other operations, such as access decisions.

Is Your Custom Role Mapping Provider Thread Safe?

For the best performance, and by default, Weblogic Server supports parallel modification to security policy and roles during application and module deployment. For this reason, deployable authorization and role mapping providers configured in the security realm should support parallel calls. The WebLogic deployable XACML Authorization and Role Mapping providers meet this requirement.

However, custom deployable authorization and role mapping providers may or may not support parallel calls. If your custom deployable authorization or role mapping providers do **not** support parallel calls, you need to disable the parallel security policy and role modification and instead enforce a synchronization mechanism that results in each application and module being placed in a queue and deployed sequentially.

> **Note**
>
> Enabling the synchronization mechanism affects every deployable provider configured in the realm, including the predefined WebLogic Server providers. Enabling the synchronization mechanism may negatively impact the performance of these providers.

See *Administering Security for Oracle WebLogic Server* for information on how to turn on this synchronization enforcement mechanism.

Do You Need to Develop a Custom Role Mapping Provider?

The default (that is, active) security realm for WebLogic Server includes the WebLogic Role Mapping provider and the XACML Role Mapping provider.

> **Note**
>
> The WebLogic Role Mapping provider, also referred to as the DefaultRoleMapper, is deprecated in WebLogic Server 14.1.1.0.0 and will be removed in a future release. Instead, the XACML Role Mapping provider is the default role mapping provider.

The XACML Role Mapping provider computes dynamic security roles for a specific user (subject) with respect to a specific protected WebLogic resource for each of the default users and WebLogic resources. The XACML Role Mapping provider supports the deployment and undeployment of security roles within the system. It implements XACML 2.0, the standard access control policy markup language. If you want to use a role mapping mechanism that already exists within your organization, you could create a custom role mapping provider to tie into that system.

Does Your Custom Role Mapping Provider Need to Support Application Versioning?

All authorization, role mapping, and credential mapping providers for the security realm must support application versioning in order for an application to be deployed using versions. If you develop a custom security provider for authorization, role mapping, or credential mapping and need to support versioned applications, you must implement the Versionable Application SSPI, as described in Versionable Application Providers.

How to Develop a Custom Role Mapping Provider

If the XACML Role Mapping provider does not meet your needs, you can develop a custom role mapping provider by following these steps:

1. Create Runtime Classes Using the Appropriate SSPIs, or, optionally, implement the Bulk Role Mapping Providers
2. Optionally, implement the Role Consumer SSPI
3. Generate an MBean type for your custom role mapping provider by completing the steps described in Generate an MBean Type Using the WebLogic MBeanMaker.
4. Configure the Custom Role Mapping Provider
5. Provide a Mechanism for Security Role Management

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Determine Which Provider Interface You Will Implement
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom role mapping provider by following these steps:

- Implement the RoleProvider SSPI *or* Implement the DeployableRoleProviderV2 SSPI
- Implement the RoleMapper SSPI
- Implement the SecurityRole Interface

> **Note**
>
> At least one role mapping provider in a security realm must implement the DeployableRoleProviderV2 SSPI, or else it will be impossible to deploy Web applications and EJBs.

For an example of how to create a runtime class for a custom role mapping provider, see Example: Creating the Runtime Class for the Sample Role Mapping Provider .

Implement the RoleProvider SSPI

To implement the `RoleProvider` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following method:

- getRoleMapper

  ```
  public RoleMapper getRoleMapper()
  ```

  The `getRoleMapper` method obtains the implementation of the `RoleMapper` SSPI. For a single runtime class called `MyRoleProviderImpl.java`, the implementation of the `getRoleMapper` method would be:

  ```
  return this;
  ```

  If there are two runtime classes, then the implementation of the `getRoleMapper` method could be:

  ```
  return new MyRoleMapperImpl;
  ```

This is because the runtime class that implements the `RoleProvider` SSPI is used as a factory to obtain classes that implement the `RoleMapper` SSPI.

See *Java API Reference for Oracle WebLogic Server* for the RoleProvider SSPI.

Implement the DeployableRoleProviderV2 SSPI

> **Note**
>
> The `DeployableRoleProvider` SSPI is deprecated in this release of WebLogic Server. Use the `DeployableRoleProviderV2` SSPI instead.

To implement the `DeployableRoleProviderV2` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs, Implement the RoleProvider SSPI, *and* the following methods:

- deleteApplicationRoles

  ```
  void deleteApplicationRoles(ApplicationInfo application)
  ```

  Deletes all roles for an application and is called only on the Administration Server within a WebLogic Server domain at the time an application is deleted.

- deployRole

  ```
  void deployRole(DeployRoleHandle handle, Resource resource, String roleName,
  String[] userAndGroupNames)
  ```

  Creates a role on behalf of a deployed Web application or EJB. If the role already exists, it is removed and replaced by this role.

- endDeployRoles

  ```
  void endDeployRoles(DeployRoleHandle handle)
  ```

  Marks the end of an application role deployment.

- startDeployRoles

  ```
  DeployRoleHandle startDeployRoles(ApplicationInfo application)
  ```

  Marks the beginning of an application role deployment and is called on all servers within a WebLogic Server domain where an application is targeted.

- undeployAllRoles

  ```
  void undeployAllRoles(DeployRoleHandle handle)
  ```

  Deletes a set of roles on behalf of an undeployed Web application or EJB.

See *Java API Reference for Oracle WebLogic Server* for the DeployableRoleProviderV2 SSPI.

The ApplicationInfo Interface

The ApplicationInfo interface passes data about an application deployment to a security provider. You can use this data to uniquely identity the application.

The Security Framework implements the ApplicationInfo interface for your convenience. You do not need to implement any methods for this interface.

The `DeployableAuthorizationProviderV2` and `DeployableRoleProviderV2` interfaces use `ApplicationInfo`. For example, consider an implementation of the `DeployableRoleProviderV2` methods. The Security Framework calls the `DeployableRoleProviderV2` `startDeployRoles` method and passes in the ApplicationInfo interface for this application. The ApplicationInfo data is determined based on the information supplied in the WebLogic Remote Console when an application is deployed.

The `startDeployRoles` method returns `DeployRoleHandle`, which you can then use in the other `DeployableRoleProviderV2` methods.

You use the ApplicationInfo interface to get the application identifier, the component name, and the component type for this application. Component type can be `APPLICATION`, `CONTROL_RESOURCE`, `EJB`, or `WEBAPP`, as defined in the `ApplicationInfo.ComponentType` class.

The following example shows one way to accomplish this task:

```
public DeployRoleHandle startDeployRoles(ApplicationInfo appInfo)
    throws DeployHandleCreationException
     :
// Obtain the application information...
    String appId = appInfo.getApplicationIdentifier();
    ComponentType compType = appInfo.getComponentType();
    String compName = appInfo.getComponentName();
```

The Security Framework calls the `DeployableRoleProviderV2` `deleteApplicationRoles` method and passes in the ApplicationInfo interface for this application. The `deleteApplicationRoles` method deletes all roles for an application and is called (only on the Administration Server within a WebLogic Server domain) at the time an application is deleted.

Implement the RoleMapper SSPI

To implement the `RoleMapper` SSPI, provide implementations for the following methods:

- getRoles

  ```
  public Map getRoles(Subject subject, Resource  resource, ContextHandler handler)
  ```

  The `getRoles` method returns the security roles associated with a given subject for a specified WebLogic resource, possibly using the optional information specified in the `ContextHandler`. For more information about ContextHandlers, see ContextHandlers and WebLogic Resources.

See *Java API Reference for Oracle WebLogic Server* for the RoleMapper SSPI and the `getRoles` methods.

Implement the SecurityRole Interface

The methods on the `SecurityRole` interface allow you to obtain basic information about a security role, or to compare it to another security role. These methods are designed for the convenience of security providers.

> **Note**
>
> SecurityRole implementations are returned as a Map by the getRoles() method (see Implement the RoleProvider SSPI).

To implement the `SecurityRole` interface, provide implementations for the following methods:

- equals

  ```
  public boolean equals(Object another)
  ```

  The `equals` method returns `TRUE` if the security role passed in matches the security role represented by the implementation of this interface, and `FALSE` otherwise.

- toString

  ```
  public String toString()
  ```

  The `toString` method returns this security role, represented as a String.

- hashCode

  ```
  public int hashCode()
  ```

  The `hashCode` method returns a hashcode for this security role, represented as an integer.

- getName

  ```
  public String getName()
  ```

  The `getName` method returns the name of this security role, represented as a String.

- getDescription

  ```
  public String getDescription()
  ```

  The `getDescription` method returns a description of this security role, represented as a String. The description should describe the purpose of this security role.

Example: Creating the Runtime Class for the Sample Role Mapping Provider

Example 8-1 shows the `SimpleSampleRoleMapperProviderImpl.java` class, which is the runtime class for the sample role mapping provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription` and `shutdown` (as described in Understand the Purpose of the Provider SSPIs).
- The method inherited from the `RoleProvider` SSPI: the `getRoleMapper` method (as described in Implement the RoleProvider SSPI).
- The five methods in the `DeployableRoleProviderV2` SSPI: the deleteApplicationRoles, `deployRole`, endDeployRoles, startDeployRoles, and `undeployAllRoles` methods (as described in Implement the DeployableRoleProviderV2 SSPI).
- The method in the `RoleMapper` SSPI: the `getRoles` method (as described in Implement the RoleProvider SSPI).

> **Note**
>
> The bold face code in Example 8-1 highlights the class declaration and the method signatures.

Example 8-1 SimpleSampleRoleMapperProviderImpl.java

```
package examples.security.providers.roles.simple;

import java.security.Principal;
import java.util.Collections;
import java.util.Date;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
import java.util.Properties;
import java.util.Set;
import jakarta.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.SubjectUtils;
import weblogic.security.WLSPrincipals;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.ApplicationInfo;
import weblogic.security.spi.ApplicationInfo.ComponentType;
import weblogic.security.spi.DeployableRoleProviderV2;
import weblogic.security.spi.DeployRoleHandle;
import weblogic.security.spi.Resource;
import weblogic.security.spi.RoleMapper;
import weblogic.security.spi.SecurityServices;
import weblogic.security.spi.VersionableApplicationProvider;

public final class SimpleSampleRoleMapperProviderImpl
implements DeployableRoleProviderV2, RoleMapper, VersionableApplicationProvider
{
   private String              description;
// a description of this provider
   private SimpleSampleRoleMapperDatabase database;
// manages the role definitions for this provider
   private static final Map NO_ROLES = Collections.unmodifiableMap(new HashMap(1));
// used when no roles are found

  public void initialize(ProviderMBean mbean, SecurityServices services)
  {
     System.out.println("SimpleSampleRoleMapperProviderImpl.initialize");

// Cast the mbean from a generic ProviderMBean to a SimpleSampleRoleMapperMBean.
     SimpleSampleRoleMapperMBean myMBean = (SimpleSampleRoleMapperMBean)mbean;

     // Set the description to the simple sample role mapper's mbean's description and version
     description = myMBean.getDescription() + "\n" + myMBean.getVersion();

     // Instantiate the helper that manages this provider's role definitions
     database = new SimpleSampleRoleMapperDatabase(myMBean);
  }
  public String getDescription()
{
     return description;
}

  public void shutdown()
{
     System.out.println("SimpleSampleRoleMapperProviderImpl.shutdown");
}

  public RoleMapper getRoleMapper()
```

```
  {
     // Since this class implements both the DeployableRoleProvider
     // and RoleMapper interfaces, this object is the
     // role mapper object so just return "this".
     return this;
  }

public Map getRoles(Subject subject, Resource resource, ContextHandler handler)
{
   System.out.println("SimpleSampleRoleMapperProviderImpl.getRoles");
   System.out.println("\tsubject\t= " + subject);
   System.out.println("\tresource\t= " + resource);

   // Make a list for the roles
   Map roles = new HashMap();


   // Make a list for the roles that have already been found and evaluated
   Set rolesEvaluated = new HashSet();

   // since resources scope roles, and resources are hierarchical,
   // loop over the resource and all its parents, adding in any roles
   // that match the current subject.
     for (Resource res = resource; res != null; res = res.getParentResource()) {
       getRoles(res, subject, roles, rolesEvaluated);
     }

   // try global resources too
   getRoles(null, subject, roles, rolesEvaluated);

   // special handling for no matching roles
   if (roles.isEmpty()) {
     return NO_ROLES;
   }

   // return the roles we found.
   System.out.println("\troles\t= " + roles);
   return roles;
   }

public DeployRoleHandle startDeployRoles(ApplicationInfo application)
{
   String appId = application.getApplicationIdentifier();
   String compName = application.getComponentName();
   ComponentType compType = application.getComponentType();
   DeployRoleHandle handle = new SampleDeployRoleHandle(appId,compName,compType);

   // ensure that previous roles have been removed so that
   // the most up to date deployment roles are in effect
   database.removeRolesForComponent(appId, compName, compType);

   // A null handle may be returned if needed
   return handle;

   }

public void deployRole(DeployRoleHandle handle, Resource resource,
String roleName, String[] principalNames)
{
   System.out.println("SimpleSampleRoleMapperProviderImpl.deployRole");
   System.out.println("\thandle\t\t= " + ((SampleDeployRoleHandle)handle).toString());
   System.out.println("\tresource\t\t= " + resource);
```

```
   System.out.println("\troleName\t\t= " + roleName);

   for (int i = 0; principalNames != null && i < principalNames.length; i++) {
      System.out.println("\tprincipalNames[" + i + "]\t= " + principalNames[i]);
   }
   database.setRole(resource, roleName, principalNames);
}
public void endDeployRoles(DeployRoleHandle handle)
{
database.saveRoles();
}

public void undeployAllRoles(DeployRoleHandle handle)
{
   System.out.println("SimpleSampleRoleMapperProviderImpl.undeployAllRoles");
   SampleDeployRoleHandle myHandle = (SampleDeployRoleHandle)handle;
   System.out.println("\thandle\t= " + myHandle.toString());

  // remove roles
   database.removeRolesForComponent(myHandle.getApplication(),
                                    myHandle.getComponent(),
                                    myHandle.getComponentType());
}

public void deleteApplicationRoles(ApplicationInfo application)
{
   System.out.println("SimpleSampleRoleMapperProviderImpl.deleteApplicationRoles");
   String appId = application.getApplicationIdentifier();
   System.out.println("\tapplication identifier\t= " + appId);

   // clear out roles for the application
   database.removeRolesForApplication(appId);
}


private void getRoles(Resource resource, Subject subject,
              Map roles, Set rolesEvaluated)
  {
   // loop over all the roles in our "database" for this resource
   for (Enumeration e = database.getRoles(resource); e.hasMoreElements();) {
     String role = (String)e.nextElement();

     // Only check for roles not already evaluated
     if (rolesEvaluated.contains(role)) {
       continue;
     }
     // Add the role to the evaluated list
     rolesEvaluated.add(role);

     // If any of the principals is on that role, add the role to the list.
     if (roleMatches(resource, role, subject)) {

       // Add a simple sample role mapper role instance to the list of roles.
       roles.put(role, new SimpleSampleSecurityRoleImpl(role));
     }
   }
}

private boolean roleMatches(Resource resource, String role, Subject subject)
{
   // loop over the the principals that are in this role.
   for (Enumeration e = database.getPrincipalsForRole(resource, role); e.hasMoreElements();) {
```

```
     // get the next principal in this role
     String principalWant = (String)e.nextElement();

     // see if any of the current principals match this principal
     if (subjectMatches(principalWant, subject)) {
       return true;
     }
   }
return false;
}

private boolean subjectMatches(String principalWant, Subject subject)
{
   // first, see if it's a group name match
   if (SubjectUtils.isUserInGroup(subject, principalWant)) {
     return true;
   }
   // second, see if it's a user name match
   if (principalWant.equals(SubjectUtils.getUsername(subject))) {
     return true;
   }
   // didn't match
   return false;
}

public void createApplicationVersion(String appId, String sourceAppId)
{
   System.out.println("SimpleSampleRoleMapperProviderImpl.createApplicationVersion");
   System.out.println("\tapplication identifier\t= " + appId);
   System.out.println("\tsource app identifier\t= " + ((sourceAppId != null) ? sourceAppId : "None"));

   // create new roles when existing application is specified
   if (sourceAppId != null) {
     database.cloneRolesForApplication(sourceAppId,appId);
   }
}


public void deleteApplicationVersion(String appId)
{
   System.out.println("SimpleSampleRoleMapperProviderImpl.deleteApplicationVersion");
   System.out.println("\tapplication identifier\t= " + appId);

   // clear out roles for the application
   database.removeRolesForApplication(appId);
}

public void deleteApplication(String appName)
{
   System.out.println("SimpleSampleRoleMapperProviderImpl.deleteApplication");
   System.out.println("\tapplication name\t= " + appName);

   // clear out roles for the application
   database.removeRolesForApplication(appName);
}

class SampleDeployRoleHandle implements DeployRoleHandle
{
   Date date;
   String application;
   String component;
```

```
   ComponentType componentType;

   SampleDeployRoleHandle(String app, String comp, ComponentType type)
   {
     this.application = app;
     this.component = comp;
     this.componentType = type;
     this.date = new Date();
   }

   public String getApplication() { return application; }
   public String getComponent() { return component; }
   public ComponentType getComponentType() { return componentType; }

   public String toString()
{
     String name = component;
     if (componentType == ComponentType.APPLICATION)
       name = application;
     return componentType +" "+ name +" ["+ date.toString() +"]";
   }
  }
}
```

Example 8-2 shows the sample `SecurityRole` implementation that is used along with the `SimpleSampleRoleMapperProviderImpl.java` runtime class.

Example 8-2 SimpleSampleSecurityRoleImpl.java

```
package examples.security.providers.roles.simple;
import weblogic.security.service.SecurityRole;
/*package*/ class SimpleSampleSecurityRoleImpl implements SecurityRole
{
   private String roleName; // the role's name
   private int    hashCode; // the role's hash code
/*package*/ SimpleSampleSecurityRoleImpl(String roleName)
{
   this.roleName = roleName;
   this.hashCode = roleName.hashCode() + 17;
}
public boolean equals(Object genericRole)
{
   // if the other role is null, we're not the same
   if (genericRole == null) {
   return false;
   }
// if we're the same java object, we're the same
if (this == genericRole) {
   return true;
}

// if the other role is not a simple sample role mapper role,
// we're not the same
if (!(genericRole instanceof SimpleSampleSecurityRoleImpl)) {
return false;
}

// Cast the other role to a simple sample role mapper role.
SimpleSampleSecurityRoleImpl sampleRole =
(SimpleSampleSecurityRoleImpl)genericRole;
// if our names don't match, we're not the same
if (!roleName.equals(sampleRole.getName())) {
```

```
   return false;
}
// we're the same
   return true;
}
public String toString()
{
return roleName;
}

public int hashCode()
{
return hashCode;
}

public String getName()
{
   return roleName;
}
public String getDescription()
{
   return "";
}
}
```

Role Consumer SSPI

WebLogic Server implements a role consumer for Web service annotations. This release of WebLogic Server includes an SSPI that role mapping providers can use to obtain the role collections.

The `RoleConsumer` SSPI is optional; only those role mapping providers that implement the SSPI are called to consume a role collection.

The SSPI supports both the delivery of initial role collections and the delivery of updated role collections.

All role mapping providers that support the `RoleConsumer` SSPI are called to consume a role collection. Each role mapping provider can choose to skip or obtain the role collection for a given role set. In the case where a provider persists roles, the provider need only collect the role once. However, providers keeping roles in memory can obtain the role collection again.

The out-of-the-box WebLogic Server Role Mapping providers persist the role into LDAP.

Required SSPI Interfaces

If you want your custom role mapping provider to support the delivery of role collections, you must implement three interfaces:

- `weblogic.security.spi.RoleConsumerFactory`
- `weblogic.security.spi.RoleConsumer`
- `weblogic.security.spi.RoleCollectionHandler`

These interfaces are described in the sections that follow.

Implement the RoleConsumerFactory SSPI Interface

A role mapping provider implements the `RoleConsumerFactory` interface so that an instance of a `RoleConsumer` is available to the WebLogic Security Framework. The WebLogic Security Framework calls your `RoleConsumerFactory` implementation to obtain the provider's implementation of the role consumer.

The `RoleConsumerFactory` SSPI has one method, which returns your implementation of the `RoleConsumer` SSPI interface.

```
public interface RoleConsumerFactory
{
  /**
   * Obtain the implementation of the RoleConsumer
   * security service provider interface (SSPI).<P>
   *
   * @return a RoleConsumer SSPI implementation.<P>
   */
  public RoleConsumer getRoleConsumer();
}
```

Implement the RoleConsumer SSPI Interface

The `RoleConsumer` SSPI returns a role collection handler for consumption of a role collection. It has one method, `getRoleCollectionHandler()`, which takes a `RoleCollectionInfo` implementation as an argument and returns your implementation of the `RoleCollectionHandler` interface.

```
public interface RoleConsumer
{
  /**
   * Obtain a role handler for consumption of a role collection.
   *
   * @param info the RoleCollectionInfo for the role collection.
   *
   * @return a RoleCollectionHandler or NULL which indicates
   *         that the role collection is not needed.
   *
   * @exception ConsumptionException if an error occurs
   *            obtaining the handler and the role collection cannot be consumed.
   */
  public RoleCollectionHandler getRoleCollectionHandler(
                                        RoleCollectionInfo info)
    throws ConsumptionException;
}
```

The WebLogic Security Framework calls the `getRoleCollectionHandler()` method and passes data about a role collection to a security provider as an implementation of the `RoleCollectionInfo` interface. (This interface implementation is provided for you, you do not have to implement it.)

You use the `RoleCollectionInfo` `getName()`, `getVersion()`, `getTimestamp()`, and `getResourceTypes()` methods to discover information about this role collection. You then return a `RoleCollectionHandler`, or NULL to indicate that the role collection is not needed.

```
public interface RoleCollectionInfo
{
  /**
   * Get the name of the collection.
```

```
   */
  public String getName();

  /**
   * Get the runtime version of the role.
   */
  public String getVersion();

  /**
   * Get the timestamp of the role.
   */
  public String getTimestamp();

  /**
   * Get the resource types used in the role collection.
   */
  public Resource[] getResouceTypes();
}
```

Implement the RoleCollectionHandler SSPI Interface

The `RoleConsumer.getRoleCollectionHandler()` method returns your implementation of the `RoleCollectionHandler` interface. `RoleCollectionHandler` has two methods: `setRole()` and `done()`. The `setRole()` method takes a resource, a role name, and an array of user and group names that defines what user names and group names are to be assigned to that role for the given resource.

The `done()` method signals the completion of the role collection.

```
public interface RoleCollectionHandler
{
  /**
   * Set a role for the specified resource.
   */
  public void setRole(Resource resource, String roleName, String[] userAndGroupNames)
     throws ConsumptionException;


  /**
   * Signals the completion of the role collection.
   */
  public void done()
     throws ConsumptionException;

}
```

Supporting an Updated Role Collection

To support the delivery of an updated role collection, all role mapping providers that support the `RoleConsumer` SSPI need to examine the contents of the `RoleCollectionInfo` passed in the `RoleConsumer.getRoleCollectionHandler()` method to determine if a role collection has changed. Each provider must decide (possibly by configuration) how to perform conflict resolution with the initial role collection and any customized role received outside of the SSPI.

For the WebLogic Server supplied role mapping providers, customized roles will not be replaced by the updated role collection: all roles from the initial role collection will be removed and only the customized roles, plus the updated role collection, will be in effect. If the role collection info has a different timestamp or version, it's treated as an updated role collection. The collection name is used as a persistence key.

The RoleConsumerMBean

Role mapping providers that implement the Role Consumer SSPI must also implement the `weblogic.management.security.authorization.RoleConsumerMBean` to indicate that the provider supports policy consumption.

PolicyStoreMBean

This release of WebLogic Server includes support for a new MBean (`weblogic.management.security.authorization.PolicyStoreMBean`) that allows for standard management (add, delete, get, list, modify, read) of administrator-generated XACML policies and policy sets. An authorization or role mapping provider MBean can optionally implement this MBean interface.

The PolicyStoreMBean methods allow security administrators to manage policy in the server as XACML documents. This includes creating and managing a domain that uses the default XACML provider, as well as managing XACML documents that the administrator has created. The administrator can then use WLST to manage these XACML policies in WebLogic Server.

WebLogic Server includes an implementation of this MBean for use with the out-of-the-box XACML providers, and you can write your own implementation of this MBean for use with your own custom authorization or role mapping providers. The WebLogic Server out-of-the-box XACML providers support the mandatory features of XACML, as described in the XACML 2.0 Core Specification (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf), with the Oracle-specific usage described in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Policies are expressed as XACML 2.0 Policy or PolicySet documents. Custom authorization providers should expect standard Policy or PolicySet documents as described in the XACML 2.0 Core Specification. Custom role mapping providers should expect Policy or PolicySet documents consistent with role assignment policies described by the Core and hierarchical role based access control (RBAC) profile of XACML v2.0 (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf).

Specifically, the Target must contain:

- An `ActionAttributeDesignator` with the id, `urn:oasis:names:tc:xacml:1.0:action:action-id`, and the value, `urn:oasis:names:tc:xacml:2.0:actions:enableRole`, according to `anyURI-equal`. For example:

```
<Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal">

<AttributeValue
DataType="http://www.w3.org/2001/
XMLSchema#anyURI">urn:oasis:names:tc:xacml:2.0:actions:enableRole
</AttributeValue>

<ActionAttributeDesignator
AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="true"/>

</ActionMatch>
</Action>
```

- A `ResourceAttributeDesignator` with the id, `urn:oasis:names:tc:xacml:2.0:subject:role`, and a value naming the role being assigned, according to string-equal. For example:

```
<ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0:resource:resource-
ancestor-or-self"
DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"/>
```

Examining the Format of a XACML Policy File

The XACML 2.0 Core Specification (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf) and the Oracle extensions described in *Securing Resources Using Roles and Policies for Oracle WebLogic Server* are the definitive sources of information for the XACML policy files used by the supplied XACML Authorization and Role Mapping Providers.

However, as part of your development process, you may want to take a look at the format of a supported XACML file. Use WLST to export the data from the XACML Authorization or Role Mapping provider's database as a XACML file. Copy this exported XACML file to a file with some other name and use the tool of your choice to review the copy.

> **Note**
>
> Treat the exported file as read-only. If you do make changes, do not import the file back into WebLogic Server. Editing exported files might result in an unusable WebLogic Server configuration and is not supported.

Using WLST to Add a Policy to the PolicyStoreMBean

Example 8-3 shows an example of using WLST to add a single policy to an instance of the PolicyStoreMBean from a XACML file.

The example assumes that you have defined the properties used in this script elsewhere, in a manner similar to the following lines from an `ant` script:

```
<property name="xacml-docs-dir" value="${xacmldir}/xacml-docs"/>
<sysproperty key="file" value="${xacml-docs-dir}/policy-getSubject.xacml"/>
```

You should avoid entering clear-text passwords in WLST commands in general, and you should especially avoid saving on disk WLST scripts that include clear-text passwords. In these instances you should use a mechanism for passing encrypted passwords instead. See Security for WLST in *Understanding the WebLogic Scripting Tool*.

Example 8-3 Using WLST to Add a Policy to the PolicyStoreMBean

```
:
try:
      protocol = System.getProperty("protocol")
      host = System.getProperty("host")
      user = System.getProperty("authuser")
      passwd = System.getProperty("authpwd")
      port = System.getProperty("port")
      dom = System.getProperty("domain")
      rlm = System.getProperty("realm")
      fil = System.getProperty("file")
      prov = System.getProperty("provider")
      stat = System.getProperty("status")
```

```
def configure():
try:
      url = protocol + "://" + host + ":" + port
      connect(user,passwd, url)
      path = "/SecurityConfiguration/" + dom + "/Realms/" + rlm + "/" + prov
      print("cd'ing to " + path)
      cd(path)
      print("calling open()")
      xacmlFile = open(fil,"r")
      print("calling read()")
      xacmlDoc = xacmlFile.read()
      print("calling cmo.addPolicy")
      if stat == "none":
          cmo.addPolicy(xacmlDoc)
      else:
          cmo.addPolicy(xacmlDoc, stat)
      print("Add error handling")
:
:
```

As described in the Navigating and Interrogating MBeans section of *Understanding the WebLogic Scripting Tool*, when WLST first connects to an instance of WebLogic Server, the variable, `cmo` (Current Management Object), is initialized to the root of all configuration management objects, `DomainMBean`. When you navigate to an MBean type, in this case `SecurityConfigurationMBean`, the value of `cmo` reflects `SecurityConfigurationMBean`. When you navigate to an MBean instance, in this case to an Authorizer MBean that implements the PolicyStoreMBean, identified in the example by the variable `prov`, WLST changes the value of `cmo` to be the current MBean instance.

The example uses the `addPolicy()` method of the PolicyStoreMBean to add a policy read from a XACML file to the policy store. Two variants of the `addPolicy()` method (without and with status) are shown.

If you use an `addPolicy()` method that does not specify status, it defaults to ACTIVE, which indicates that the policy is evaluated for any decision to which its target applies. You can explicitly set status to be ACTIVE, INACTIVE, or BYREFERENCE. The INACTIVE status indicates that the policy will never be evaluated and is only being stored. The BYREFERENCE status indicates that the policy will only be evaluated when referenced by a policy set that is being evaluated.

You can invoke this type of WLST script from the command line, in a manner similar to the following:

```
java -Dhost="localhost " -Dprotocol="t3" -Dauthuser="weblogic"
-Dauthpwd="weblogic" -Dport="7001" -Ddomain="mydomain" -Drealm="myrealm"
-Dprovider="Authorizers/XACMLAuthorizer"
-Dfile="C:/XACML/xacml-docs/policy12.xml" -Dstatus="none" weblogic.WLST
XACML/scripts/XACMLaddPolicy.py
```

Using WLST to Read a PolicySet as a String

Example 8-4 shows an example of using WLST to read a PolicySet as a string.

The example assumes that you have defined the properties used in this script elsewhere, in a manner similar to the following lines from an `ant` script:

```
<sysproperty key="identifier"
value="urn:sample:xacml:2.0:wlssecqa:resource:type@E@Fejb@G@M@Oapplication@ENoD
DRolesOrPoliciesEar@M@Omodule@Eejb11inEarMiniAppBean.jar@M@Oejb@EMiniAppBean@
```

```
M@Omethod@EgetSubject@M@OmethodInterface@ERemote"/>
<sysproperty key="version" value="1.0"/>
```

You should avoid entering clear-text passwords in WLST commands in general, and you should especially avoid saving on disk WLST scripts that include clear-text passwords. In these instances you should use a mechanism for passing encrypted passwords instead. See Security for WLST in *Understanding the WebLogic Scripting Tool*.

Example 8-4 Using WLST to Read a PolicySet as a String

```
:
:
try:
      print("start XACMLreadPolicySet.py")
      protocol = System.getProperty("protocol")
      host = System.getProperty("host")
      user = System.getProperty("authuser")
      passwd = System.getProperty("authpwd")
      port = System.getProperty("port")
      dom = System.getProperty("domain")
      rlm = System.getProperty("realm")
      prov = System.getProperty("provider")
      id = System.getProperty("identifier")
      vers = System.getProperty("version")
:
:
def configure():
try:
      url = protocol + "://" + host + ":" + port
      connect(user,passwd, url)
      path = "/SecurityConfiguration/" + dom + "/Realms/" + rlm + "/" + prov
      print("cd'ing to " + path)
      cd(path)
      polset = cmo.readPolicySetAsString(id, vers)
      print("readPolicySetAsString() returned the following policy set: " + polset)
      print"Add error handling."
:
:
```

As described in the XACML 2.0 Core Specification (http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf), the <PolicySet> element contains a set of <Policy> or other <PolicySet> elements and a specified procedure for combining the results of their evaluation. See the XACML 2.0 Core Specification for complete information.

Bulk Role Mapping Providers

This release of WebLogic Server includes bulk access versions of the following role mapping provider SSPI interfaces:

- `BulkRoleProvider`
- `BulkRoleMapper`

The bulk access SSPI interfaces allow role mapping providers to receive multiple decision requests in one call rather than through multiple calls, typically in a 'for' loop. The intent of the bulk SSPI variants is to allow provider implementations to take advantage of internal performance optimizations, such as detecting that many of the passed-in `Resource` objects are protected by the same policy and will generate the same decision result.

There are subtle differences in how the non-bulk and bulk versions of the SSPI interfaces are used. For example, the `BulkRoleMapper.getRoles()` method returns a `Map` of roles indexed

first by resource and then by their names (`Map<Resource, Map<String, SecurityRole>>`), representing the security roles associated with the specified resources that have been granted to the subject.

Configure the Custom Role Mapping Provider

Configuring a custom role mapping provider means that you are adding the custom role mapping provider to your security realm, where it can be accessed by applications requiring role mapping services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers. This section contains information that is important for the person configuring your custom role mapping providers:

- Managing Role Mapping Providers and Deployment Descriptors
- Enabling Security Role Deployment

> **Note**
>
> The steps for configuring a custom role mapping provider are described in Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

Managing Role Mapping Providers and Deployment Descriptors

Some application components, such as Enterprise JavaBeans (EJBs) and Web applications, store relevant deployment information in Java EE and WebLogic Server deployment descriptors. For Web applications, the deployment descriptor files (called `web.xml` and `weblogic.xml`) contain information for implementing the Java EE security model, including security roles. Typically, you will want to include this information when first configuring your role mapping providers in the WebLogic Remote Console.

Because the Java EE platform standardizes Web application and EJB security in deployment descriptors, WebLogic Server integrates this standard mechanism with its Security Service to give you a choice of techniques for securing Web application and EJB resources. You can use deployment descriptors exclusively, the WebLogic Remote Console exclusively, or you can combine the techniques for certain situations.

Depending on the technique you choose, you also need to apply a Security Model. WebLogic supports different security models for individual deployments, and a security model for realm-wide configurations that incorporate the technique you want to use.

See Options for Securing EJB and Web Application Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

When configured to use deployment descriptors, WebLogic Server reads security role information from the `web.xml` and `weblogic.xml` deployment descriptor files (examples of `web.xml` and `weblogic.xml` files are shown in Example 8-5 and Example 8-6. This information is then copied into the security provider database for the role mapping provider.

Example 8-5 Sample web.xml File

```
<web-app>
   <welcome-file-list>
      <welcome-file>welcome.jsp</welcome-file>
   </welcome-file-list>
```

```
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Success</web-resource-name>
         <url-pattern>/welcome.jsp</url-pattern>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
         <role-name>developers</role-name>
      </auth-constraint>
   </security-constraint>
   <login-config>
       <auth-method>BASIC</auth-method>
       <realm-name>default</realm-name>
   </login-config>
   <security-role>
      <role-name>developers</role-name>
   </security-role>
</web-app>
```

Example 8-6 Sample weblogic.xml File

```
<weblogic-web-app>
   <security-role-assignment>
      <role-name>developers</role-name>
      <principal-name>myGroup</principal-name>
   </security-role-assignment>
</weblogic-web-app>
```

Enabling Security Role Deployment

If you implemented the `DeployableRoleProviderV2` SSPI as part of developing your custom role mapping provider and want to support deployable security roles, the person configuring the custom role mapping provider (that is, you or an administrator) must be sure that the Role Deployment Enabled box in the WebLogic Remote Console is checked. Otherwise, deployment for the role mapping provider is considered turned off. Therefore, if multiple role mapping providers are configured, the Role Deployment Enabled box can be used to control which role mapping provider is used for security role deployment.

Provide a Mechanism for Security Role Management

While configuring a custom role mapping provider using the WebLogic Remote Console makes it accessible by applications requiring role mapping services, you also need to supply administrators with a way to manage this security provider's associated security roles. The WebLogic Role Mapping provider, for example, supplies administrators with a Role Editor page that allows them to add, modify, or remove security roles for various WebLogic resources.

Neither the Role Editor page nor access to it is available to administrators when you develop a custom role mapping provider. Therefore, you must provide your own mechanism for security role management. This mechanism must read and write security role data (that is, expressions) to and from the custom role mapping provider's database.

For more information, see Develop a Stand-Alone Tool for Security Role Management.

Develop a Stand-Alone Tool for Security Role Management

You would typically select this option if you want to develop a tool that is entirely separate from the WebLogic Remote Console.

For this option, you do not need to develop any management MBeans. However, your tool needs to:

1. Determine the WebLogic resource's ID. For more information, see WebLogic Resource Identifiers.
2. Determine how to represent the expressions that make up a security role. (This representation is entirely up to you and need not be a string.)
3. Read and write the expressions from and to the custom role mapping provider's database.

9

Auditing Providers

This chapter describes Auditing provider concepts and functionality, and provides step-by-step instructions for developing a custom Auditing provider.
Auditing is the process whereby information about operating requests and the outcome of those requests are collected, stored, and distributed for the purposes of non-repudiation. In WebLogic Server, an Auditing provider provides this electronic trail of computer activity.

This chapter includes the following sections:

Auditing Concepts

Before you develop an Auditing provider, you need to understand the following concepts:

Audit Channels

An **Audit Channel** is the component of an Auditing provider that determines whether a security event should be audited, and performs the actual recording of audit information based on Quality of Service (QoS) policies.

> **Note**
>
> See Implement the AuditChannel SSPI.

Auditing Events From Custom Security Providers

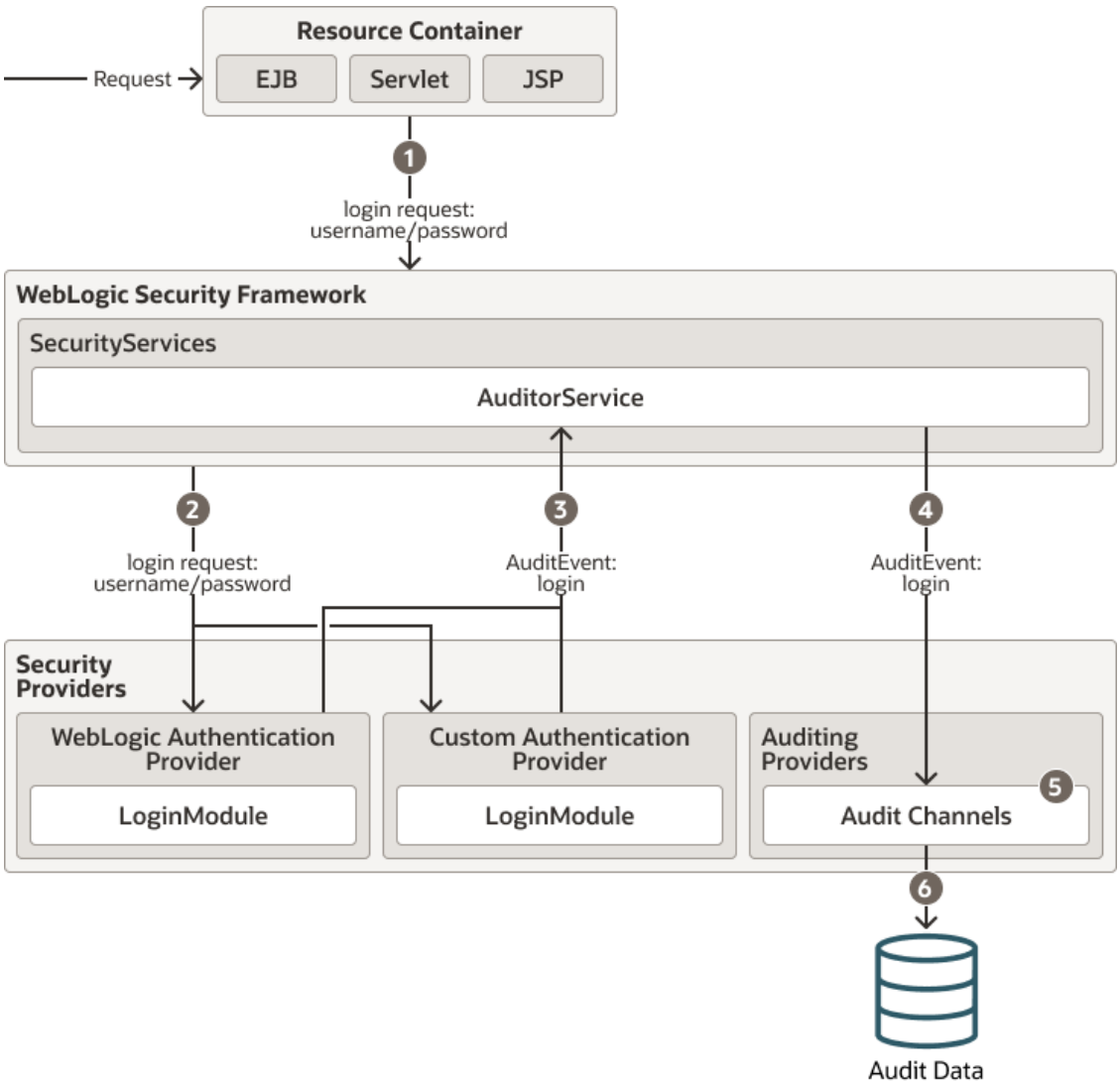
Each type of security provider can call the configured Auditing providers with a request to write out information about security-related events, before or after these events take place. For example, if a user attempts to access a withdraw method in a bank account application (to which they should not have access), the authorization provider can request that this operation be recorded. Security-related events are only recorded when they meet or exceed the severity level specified in the configuration of the Auditing providers.

For information about how to post audit events from a custom security provider, see Auditing Events From Custom Security Providers.

The Auditing Process

Figure 9-1 shows how Auditing providers interact with the WebLogic Security Framework and other types of security providers (using authentication providers as an example) to audit selected events. An explanation follows.

Figure 9-1 Auditing Providers, the WebLogic Security Framework, and Other Security Providers



Auditing providers interact with the WebLogic Security Framework and other types of security providers in the following manner:

> **Note**
>
> In Figure 9-1 and the explanation below, the other types of security providers are a WebLogic Authentication provider and a custom authentication provider. However, these can be any type of security provider that is developed as described in Auditing Events From Custom Security Providers.

1. A resource container passes a user's authentication information (for example, a username/password combination) to the WebLogic Security Framework as part of a login request.
2. The WebLogic Security Framework passes the information associated with the login request to the configured authentication providers.
3. If, in addition to providing authentication services, the authentication providers are designed to post audit events, the authentication providers will each:
   a. Instantiate an `AuditEvent` object. At minimum, the `AuditEvent` object includes information about the event type to be audited and an audit severity level.

      > **Note**
      >
      > An AuditEvent class is created by implementing either the AuditEvent SSPI or an AuditEvent convenience interface in the authentication provider's runtime class, in addition to the other security service provider interfaces (SSPIs) the custom authentication provider must already implement. See Create an Audit Event.

   b. Make a trusted call to the Auditor Service, passing in the `AuditEvent` object.

      > **Note**
      >
      > This is a trusted call because the Auditor Service is already passed to the security provider's initialize method as part of its Provider SSPI implementation. See Understand the Purpose of the Provider SSPIs.

4. The Auditor Service passes the `AuditEvent` object to the configured Auditing providers' runtime classes (that is, the `AuditChannel` SSPI implementations), enabling audit event recording.

   > **Note**
   >
   > Depending on the authentication providers' implementations of the AuditEvent convenience interface, audit requests may occur both pre and post event, as well as just once for an event.

5. The Auditing providers' runtime classes use the event type, audit severity and other information (such as the Audit Context) obtained from the `AuditEvent` object to control audit record content. Typically, only one of the configured Auditing providers will meet all the criteria for auditing.

> **Note**
>
> See Audit Severity and Audit Context, respectively.

6. When the criteria for auditing specified by the authentication providers in their `AuditEvent` objects is met, the appropriate Auditing provider's runtime class (that is, the `AuditChannel` SSPI implementation) writes out audit records in the manner their implementation specifies.

> **Note**
>
> Depending on the `AuditChannel` SSPI implementation, audit records may be written to a file, a database, or some other persistent storage medium when the criteria for auditing is met.

Implementing the ContextHandler MBean

The ContextHandlerMBean, `weblogic.management.security.audit.ContextHandler`, provides a set of attributes for ContextHandler support. You use this interface to manage audit providers that support context handler entries in a standard way.

An Auditor provider MBean can optionally implement the `ContextHandlerMBean` MBean. The Auditor provider can then use the MBean to determine the supported and active ContextHandler entries.

The WebLogic Remote Console detects when an Auditor provider implements this MBean and automatically provides a tab for using these attributes.

> **Note**
>
> The ContextHandler entries associated with the ContextHandlerMBean are not related to, nor do they affect, the contents of an AuditEvent that is passed to the Audit providers. An AuditEvent received by a provider may or may not include a ContextHandler with ContextElements. If a ContextHandler is included, an Audit provider can get the ContextHandler from the AuditEvent, regardless of whether you implemented the ContextHandlerMBean management interface. In particular, the AuditContext getContext method returns a weblogic.security.service.ContextHandler interface that is independent of the context handler implemented by the ContextHandlerMBean.
>
> You can choose to implement the ContextHandlerMBean context handler in a manner that compliments the AuditContext getContext method. (The `SimpleSampleAuditProviderImpl.java` sample takes this approach.) However, there is no requirement that you do so.

ContextHandlerMBean Methods

The `ContextHandlerMBean` interface implements the following methods:

- getActiveContextHandlerEntries

```
public String[] getActiveContextHandlerEntries()
```

Returns the ContextHandler entries that the Audit provider is currently configured to process.

- getSupportedContextHandlerEntries

```
public String[] getSupportedContextHandlerEntries()
```

Returns the list of all ContextHandler entries supported by the auditor.

- setActiveContextHandlerEntries

```
public void setActiveContextHandlerEntries(String[] types) throws
InvalidAttributeValueException
```

Sets the ContextHandler entries that the Audit provider will process. The entries you specify must be listed in the Audit provider's SupportedContextHandlerEntries attribute.

Example: Implementing the ContextHandlerMBean

Example 9-5 shows the `SimpleSampleAuditProviderImpl.java` class, which is the runtime class for the sample Auditing provider. This sample Auditing provider has been enhanced to implement the `ContextHandlerMBean`.

An MBean Definition File (MDF) is an XML file used by the WebLogic MBeanMaker utility to generate the Java files that comprise an MBean type. All MDFs must extend a required SSPI MBean that is specific to the type of the security provider you have created, and can implement optional SSPI MBeans.

Example 9-1 shows the key sections of the MDF for the sample Auditing provider, which implements the optional ContexthandlerMBean.

Example 9-1 Example: SimpleSampleAuditor.xml

```
<MBeanType
Name          = "SimpleSampleAuditor"
DisplayName   = "SimpleSampleAuditor"
Package       = "examples.security.providers.audit.simple"
Extends       = "weblogic.management.security.audit.Auditor"
Implements    = "weblogic.management.security.audit.ContextHandler"
PersistPolicy = "OnUpdate"
>
...
<MBeanAttribute
Name          = "SupportedContextHandlerEntries"
Type          = "java.lang.String[]"
Writeable     = "false"
Default       = "new String[] {
&quot;com.bea.contextelement.servlet.HttpServletRequest&quot; }"
Description   = "List of all ContextHandler entries
supported by the auditor."
/>
```

Extend weblogic.management.security.audit.ContextHandlerImpl

The ContextHandlerMBean has an `setActiveContextHandlerEntries` attribute that sets the ContextHandler entries that the Audit provider is currently configured to process. The entries you specify must be listed in the Audit provider's `SupportedContextHandlerEntries` attribute. However, this requirement is not actually enforced by the MBean. Additional work is required to

validate that this attribute can set only values from the `SupportedContextHandlerEntries` attribute.

You must also create an MBean customizer (for example, you might call it `MyAuditorImpl.java`) file that extends `weblogic.management.security.audit.ContextHandlerImpl`. Extending `weblogic.management.security.audit.ContextHandlerImpl` gives the provider access to the `ActiveContextHandlerEntries` attribute validator, which ensures that the entries include only `SupportedContextHandlerEntries`.

An example of extending `ContextHandlerImpl` is available in `SimpleSampleAuditorImpl`, which is shown in Example 9-2.

After you implement code similar to that in `SimpleSampleAuditorImpl`, add code to your Audit runtime provider to get the `ActiveContextHandlerEntries`. One possible way to do this is shown in Example 9-3.

Example 9-2 SimpleSampleAuditorImpl

```
package examples.security.providers.audit.simple;
import javax.management.MBeanException;
import javax.management.modelmbean.RequiredModelMBean;
import weblogic.management.security.audit.ContextHandlerImpl;

/**
* The simple sample auditor's mbean implementation.
* <p>
* It is needed to inherit the ContextHandlerMBean's ActiveContextHandlerEntries
* attribute validator that ensures that the ActiveContextHandlerEntries
* attribute only contains values from the SupportedContextHandlerEntries
* attribute.
*
* @author Copyright © 1996, 2008, Oracle and/or its affiliates.
* All rights reserved.
*/
public class SimpleSampleAuditorImpl extends ContextHandlerImpl
// Note: extend ContextHandlerImpl instead of AuditorImpl to inherit
// the ActiveContextHandlerEntries attribute validator.
{
/**
* Standard mbean impl constructor.
*
* @throws MBeanException
*/
public SimpleSampleAuditorImpl(RequiredModelMBean base) throws MBeanException
{
super(base);
}
}
```

Example 9-3 Getting Active Context Handler Entries

```
 String [] activeHandlerEntries = myMBean.getActiveContextHandlerEntries();
    if (activeHandlerEntries != null) {
      for (int i=0; i<activeHandlerEntries.length; i++) {
        if ((activeHandlerEntries[i] != null) &&
            (activeHandlerEntries[i].equalsIgnoreCase(HTTP_REQUEST_ELEMENT))) {
          handlerEnabled = true;
          break;
        }
      }
    }
```

Do You Need to Develop a Custom Auditing Provider?

The default (that is, active) security realm for WebLogic Server includes a WebLogic Auditing provider. The WebLogic Auditing provider records information from a number of security requests, which are determined internally by the WebLogic Security Framework. The WebLogic Auditing provider also records the event data associated with these security requests, and the outcome of the requests.

The WebLogic Auditing provider makes an audit decision in its `writeEvent` method, based on the audit severity level it has been configured with and the audit severity contained within the AuditEvent object that is passed into the method. See Create an Audit Event.

> **Note**
>
> You can change the audit severity level that the WebLogic Auditing provider is configured with using the WebLogic Remote Console. See Configuring a WebLogic Auditing Provider in *Administering Security for Oracle WebLogic Server*.

If there is a match, the WebLogic Auditing provider writes audit information to the `DefaultAuditRecorder.log` file, which is located in the *`WL_HOME`*`\yourdomain\yourserver\logs` directory. Example 9-4 is an excerpt from the `DefaultAuditRecorder.log` file.

Example 9-4 DefaultAuditRecorder.log File: Sample Output

```
When Authentication suceeds. [SUCCESS]
#### Audit Record Begin <Feb 23, 2005 11:42:17 AM>  <Severity=SUCCESS>
<<<Event Type = Authentication Audit Event><TestUser><AUTHENTICATE>>> Audit
Record End ####
When Authentication fails. [FAILURE]
#### Audit Record Begin <Feb 23, 2005 11:42:01 AM>  <Severity=FAILURE>
<<<Event Type = Authentication Audit Event><TestUser><AUTHENTICATE>>> Audit
Record End ####When Operations are invoked.[SUCCESS]
When a user account is unlocked. [SUCCESS]
#### Audit Record Begin <Feb 23, 2005 11:42:17 AM>  <Severity=SUCCESS>
<<<Event Type = Authentication Audit Event><TestUser><USERUNLOCKED>>> Audit
Record End ####
When an Authorization request succeeds. [SUCCESS]
#### Audit Record Begin <Feb 23, 2005 11:42:17 AM>  <Severity=SUCCESS>
<<<Event Type = Authorization Audit Event ><Subject: 1
Principal = class weblogic.security.principal.WLSUserImpl("TestUser")
><ONCE><<jndi>><type=<jndi>, application=, path={weblogic}, action=lookup>>>
Audit Record End ####
```

Specifically, Example 9-4 shows the Role Manager (a component in the WebLogic Security Framework that deals specifically with security roles) recording an audit event to indicate that an authorized administrator has accessed a protected method in a certificate servlet.

You can specify a new directory location for the `DefaultAuditRecorder.log` file on the command line with the following Java startup option:

```
-Dweblogic.security.audit.auditLogDir=c:\foo
```

The new file location will be `c:\foo\yourserver\DefaultAuditRecorder.log`.

If you want to write audit information in addition to that which is specified by the WebLogic Security Framework, or to an output repository that is not the DefaultAuditRecorder.log (that is, to a simple file with a different name/location or to an existing database), then you need to develop a custom Auditing provider.

How to Develop a Custom Auditing Provider

If the WebLogic Auditing provider does not meet your needs, you can develop a custom Auditing provider by following these steps:

1. Create Runtime Classes Using the Appropriate SSPIs
2. Generate an MBean type for your custom auditing provider by completing the steps described in Generate an MBean Type Using the WebLogic MBeanMaker.
3. Configure the Custom Auditing Provider

> **Note**
>
> After creating a custom Auditing provider, if you are using WLST to manage your custom Auditing provider configuration, you must ensure that the provider interface jar is specified in the `WLST_EXT_CLASSPATH` environment variable. Optionally, you can set the location of the directory containing the provider jar using the `-Dweblogic.alternateTypesDirectory` system property in the `CONFIG_JVM_ARGS` environment variable.

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs.
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom Auditing provider by following these steps:

- Implement the AuditProvider SSPI
- Implement the AuditChannel SSPI

For an example of how to create a runtime class for a custom Auditing provider, see Example: Creating the Runtime Class for the Sample Auditing Provider.

Implement the AuditProvider SSPI

To implement the AuditProvider SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following method:

- getAuditChannel

  ```
  public AuditChannel getAuditChannel();
  ```

  The `getAuditChannel` method obtains the implementation of the `AuditChannel` SSPI. For a single runtime class called `MyAuditProviderImpl.java`, the implementation of the `getAuditChannel` method would be:

```
return this;
```

If there are two runtime classes, then the implementation of the getAuditChannel method could be:

```
return new MyAuditChannelImpl;
```

This is because the runtime class that implements the `AuditProvider` SSPI is used as a factory to obtain classes that implement the `AuditChannel` SSPI.

See *Java API Reference for Oracle WebLogic Server* for the AuditProvider SSPI.

Implement the AuditChannel SSPI

To implement the AuditChannel SSPI, provide an implementation for the following method:

- writeEvent

  ```
  public void writeEvent(AuditEvent event)
  ```

  The `writeEvent` method writes an audit record based on the information specified in the `AuditEvent` object that is passed in. See Create an Audit Event.

See *Java API Reference for Oracle WebLogic Server* for the AuditChannel SSPI.

Example: Creating the Runtime Class for the Sample Auditing Provider

Example 9-5 shows the `SimpleSampleAuditProviderImpl.java` class, which is the runtime class for the sample Auditing provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription` and `shutdown` (as described in Understand the Purpose of the Provider SSPIs.)
- The method inherited from the `AuditProvider` SSPI: the `getAuditChannel` method (as described in Implement the AuditProvider SSPI).
- The method in the `AuditChannel` SSPI: the `writeEvent` method (as described in Implement the AuditChannel SSPI).

> **Note**
>
> The bold face code in Example 9-5 highlights the class declaration and the method signatures.

Example 9-5 SimpleSampleAuditProviderImpl.java

```
package examples.security.providers.audit.simple;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import jakarta.servlet.http.HttpServletRequest;
import weblogic.management.security.ProviderMBean;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AuditChannel;
import weblogic.security.spi.AuditContext;
import weblogic.security.spi.AuditEvent;
import weblogic.security.spi.AuditProvider;
```

```
import weblogic.security.spi.SecurityServices;

public final class SimpleSampleAuditProviderImpl implements AuditProvider, AuditChannel
{
   private String      description; // a description of this provider
   private PrintStream log;         // the log file that events are written to
   private boolean     handlerEnabled = false;
   private final static String HTTP_REQUEST_ELEMENT =
 "com.bea.contextelement.servlet.HttpServletRequest";
   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
     System.out.println("SimpleSampleAuditProviderImpl.initialize");
     SimpleSampleAuditorMBean myMBean = (SimpleSampleAuditorMBean)mbean;
     description = myMBean.getDescription() + "\n" + myMBean.getVersion();
     String [] activeHandlerEntries = myMBean.getActiveContextHandlerEntries();
     if (activeHandlerEntries != null) {
       for (int i=0; i<activeHandlerEntries.length; i++) {
         if ((activeHandlerEntries[i] != null) &&
             (activeHandlerEntries[i].equalsIgnoreCase(HTTP_REQUEST_ELEMENT))) {
           handlerEnabled = true;
           break;
        }
       }
     }
     File file = new File(myMBean.getLogFileName());
     System.out.println("\tlogging to " + file.getAbsolutePath());
     try {
       log = new PrintStream(new FileOutputStream(file), true);
     } catch (IOException e) {
       throw new RuntimeException(e.toString());
     }
   }

   public String getDescription()
   {
     return description;
   }
   public void shutdown()
   {
     System.out.println("SimpleSampleAuditProviderImpl.shutdown");
     log.close();
   }
   public AuditChannel getAuditChannel()
   {
     return this;
   }
   public void writeEvent(AuditEvent event)
   {
     log.println(event);

     if ((!handlerEnabled) || (!(event instanceof AuditContext)))
       return;

     AuditContext auditContext = (AuditContext)event;
     ContextHandler handler = auditContext.getContext();

     if ((handler == null) || (handler.size() == 0))
       return;

     Object requestValue = handler.getValue("com.bea.contextelement.servlet.HttpServletRequest");
     if ((requestValue == null) || (!(requestValue instanceof HttpServletRequest)))
       return;
```

```
    HttpServletRequest request = (HttpServletRequest) requestValue;
    log.println("    " + HTTP_REQUEST_ELEMENT + " method: " + request.getMethod());
    log.println("    " + HTTP_REQUEST_ELEMENT + " URL: " + request.getRequestURL());
    log.println("    " + HTTP_REQUEST_ELEMENT + " URI: " + request.getRequestURI());
    return;
  }
}
```

Configure the Custom Auditing Provider

Configuring a custom Auditing provider means that you are adding the custom Auditing provider to your security realm, where it can be accessed by security providers requiring audit services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers. This section contains information that is important for the person configuring your custom Auditing providers:

- Configuring Audit Severity

> **Note**
>
> The steps for configuring a custom Auditing provider are described in Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

Configuring Audit Severity

During the configuration process, an Auditing provider's audit severity must be set to one of the following severity levels:

- `INFORMATION`
- `WARNING`
- `ERROR`
- `SUCCESS`
- `FAILURE`

Security Framework Audit Events

This section describes the audit events that are posted by the WebLogic Server Security Framework. If you write a custom audit provider, it should be prepared to handle these events. The following topics are covered in this section:

- Passing Additional Audit Information
- Audit Event Interfaces and Audit Events

Passing Additional Audit Information

The WebLogic Security providers implement the appropriate AuditEvent interfaces and post those events to the Audit provider. The audit events that also implement the `AuditContext` interface can provide more information via a ContextHandler.

Table 9-1 lists the `weblogic.security.spi` subinterfaces that extend the AuditEvent SSPI, and indicates which subinterfaces implement the `AuditContext` interface.

Table 9-1 Audit Events

Audit Event Name	Interface Class	Audit Event	Audit Context
Application Version Event	`weblogic.security.spi.AuditApplicationVersionEvent`	Yes	No
Authentication Audit Event	`weblogic.security.spi.AuditAtnEvent`	Yes	No
Authentication Audit Event V2	`weblogic.security.spi.AuditAtnEventV2`	Yes	Yes
Authorization Audit Event	`weblogic.security.spi.AuditAtzEvent`	Yes	Yes
CertPathBuilder Audit Event	`weblogic.security.spi.AuditCertPathBuilderEvent`	Yes	Yes
CertPathValidator Audit Event	`weblogic.security.spi.AuditCertPathValidatorEvent`	Yes	Yes
Configuration Audit Event	`weblogic.security.spi.AuditConfigurationEvent`	Yes	Yes
Credential Mapping Audit Event	`weblogic.security.spi.AuditCredentialMappingEvent`	Yes	Yes
Life Cycle Event	`weblogic.security.spi.AuditLifecycleEvent`	Yes	No
Audit Management Event	`weblogic.security.spi.AuditMgmtEvent`	Yes	No
Policy Audit Event	`weblogic.security.spi.AuditPolicyEvent`	Yes	No
Policy Consumer Audit Event	`weblogic.security.service.internal.PolicyConsumerAuditEvent`	AuditPolicyEvent	No
Provider Audit Record	`com.bea.security.spi.ProviderAuditRecord`	Yes	Yes
Role Consumer Audit Event	`weblogic.security.service.internal.RoleConsumerAuditEvent`	AuditRoleEvent	Yes
Role Deployment Audit Event	`weblogic.security.spi.AuditRoleDeploymentEvent`	Yes	No
Role Mapping Audit Event	`weblogic.security.spi.AuditRoleEvent`	Yes	Yes

Audit Event Interfaces and Audit Events

In the weblogic.security.spi package, WebLogic Security defines one top-level base interface (AuditEvent) with derived interfaces that represent the different types of audit events.

Subsequent sections describe when the security framework and security providers post the following audit events:

- `AuditApplicationVersionEvent`

- `AuditAtnEventV2`
- `AuditAtzEvent`
- `AuditCerPathBuilderEvent, AuditCertPathValidatorEvent`
- `AuditConfigurationEvent (AuditCreateConfigurationEvent, AuditDeleteConfigurationEvent, AuditInvokeConfigurationEvent, AuditSetAttributeConfigurationEvent)`
- `AuditCredentialMappingEvent`
- `AuditLifecycleEvent`
- `AuditMgmtEvent`
- `AuditPolicyEvent (AuditEndPolicyDeployEvent, AuditPolicyDeleteAppEvent, AuditPolicyDeployEvent, AuditPolicyUndeployEvent, AuditResourceProtectedEvent, AuditStartPolicyDeployEvent, PolicyConsumerAuditEvent)`
- `AuditRoleDeploymentEvent (AuditStartRoleDeployEvent, AuditEndRoleDeployEvent, AuditRoleUndeployEvent, AuditRoleDeleteAppEvent)`
- `AuditRoleEvent (RoleConsumerAuditEvent)`

AuditApplicationVersionEvent

Application version audit events are posted by the security framework. You can use the `getEventType` method to get the type of the audit event. The actual audit string returned by `getEventType` is `String = "Application Version Audit Event"`.

Table 9-2 describes the conditions under which the event is posted and severity level of the event.

Table 9-2 Application Version Events

Component	Description	Severity
Security Framework	The security framework posts these events for the following reasons: • Authorization Manager application version creation has succeeded or failed. • Authorization Manager application version deletion has succeeded or failed. • Authorization Manager non-versioned application deletion has succeeded or failed. • Role Manager application version creation has succeeded or failed. • Role Manager application version deletion has succeeded or failed. • Role Manager non-versioned application deletion has succeeded or failed. • Credential Manager application version creation has succeeded or failed. • Credential Manager application version deletion has succeeded or failed. • Credential Manager non-versioned application deletion has succeeded or failed.	Success or Failure

AuditAtnEventV2

Authentication audit events are posted by the security framework. You can use the `getEventType` method to get the type of the audit event. The actual audit string returned by `getEventType` is `String eventType = "Event Type = Authentication Audit Event"`.

Table 9-3 describes the conditions under which the event is posted and severity level of the event.

Table 9-3 Authentication Audit Events

Component	Description	Severity
Security Framework	Posted after successful authentication of a user.	Success
Security Framework	Posted after unsuccessful authentication (a LoginException thrown from JAAS login method). This LoginException can be thrown by either JAAS framework or by JAAS LoginModule of WebLogic Server authentication provider.	Failure
Security Framework	Posted after an identity assertion to an anonymous user.	Success
Security Framework	Posted after an unsuccessful identity assertion (IdentityAssertionException thrown from identity assertion method)	Failure
Security Framework	Posted after an unsuccessful identity assertion (IOException is thrown by identity assertion callback handler when retrieving username from callback).	Failure
Security Framework	Posted after an unsuccessful identity assertion (UnsupportedCallbackException is thrown by identity assertion callback handler when retrieving username from callback).	Failure
Security Framework	Posted after an unsuccessful identity assertion (when username returned from identity assertion callback handler is null or zero length).	Failure
Security Framework	Posted after a successful identity assertion.	Success
Security Framework	Posted after an unsuccessful identity assertion.	Failure
Security Framework	Posted after a successful impersonate identity (anonymous identity).	Success
Security Framework	Posted after a successful impersonate identity.	Success
Security Framework	Posted after an unsuccessful impersonate identity.	Failure
Security Framework	Posted after a failure of principal validation.	Failure

AuditAtzEvent

Authorization audit events are posted by the security framework. You can use the `getEventType` method to get the type of the audit event. The actual audit string returned by `getEventType` is `String eventType = "Event Type = Authorization Audit Event V2 "`.

Table 9-4 describes the conditions under which the events are posted and severity level of the event.

Table 9-4 Authorization Audit Events

Component	Description	Severity
Security Framework	Posted if access is not allowed to resource (exception thrown by authorization provider).	Failure
Security Framework	Posted if access is allowed to resource.	Success
Security Framework	Posted if access is not allowed to resource.	Failure

AuditCerPathBuilderEvent, AuditCertPathValidatorEvent

CertPath Builder and Validation audit events are posted by the security framework. You can use the `getEventType` method to get the type of the audit event. The actual audit strings returned by `getEventType` are as follows:

- `String eventType = "Event Type = CertPathBuilder Audit Event "`
- `String eventType = "Event Type = CertPathValidator Audit Event "`

Table 9-5 describes the conditions under which the events are posted and severity level of the event.

Table 9-5 CertPath Builder and Validation Events

Component	Description	Severity
Security Framework	Posted if the Certificate Path is successfully built.	Success
Security Framework	Posted if the Certificate Path is not successfully built.	Failure
Security Framework	Posted if the Certificate Path is successfully validated.	Success
Security Framework	Posted if the Certificate Path is not successfully validated.	Failure

AuditConfigurationEvent

Configuration audit events are posted by the security framework. The following events are posted:

- `AuditConfigurationEvent`
- `AuditCreateConfigurationEvent` (The actual audit string returned by `getEventType` is `String CREATE_EVENT = "Create Configuration Audit Event"`)

- `AuditDeleteConfigurationEvent` (The actual audit string returned by `getEventType` is `String DELETE_EVENT = "Delete Configuration Audit Event"`)
- `AuditInvokeConfigurationEvent` (The actual audit string returned by `getEventType` is `String INVOKE_EVENT = "Invoke Configuration Audit Event"`)
- `AuditSetAttributeConfigurationEvent` (The actual audit string returned by `getEventType` is `String SETATTRIBUTE_EVENT = "SetAttribute Configuration Audit Event"`)

Table 9-6 describes the conditions under which the events are posted and severity level of the events.

Table 9-6 Audit Configuration Events

Component	Description	Severity
WebLogic Management Infrastructure	The WebLogic Management infrastructure implements this interface and may post configuration audit events for the following configuration change events: • A request to create a new configuration artifact has been allowed or disallowed. • A request to delete an existing configuration artifact has been allowed or disallowed. • A request to modify an existing configuration artifact has been allowed or disallowed. • A invoke an operation on an existing configuration artifact has been allowed or disallowed.	Success or Failure

AuditCredentialMappingEvent

Credential mapping audit events are posted by the security framework. You can use the `getEventType` method to get the type of the audit event. The actual audit string returned by `getEventType` is `String EVENT_TYPE = "Event Type = Credential apping Audit Event"`.

Table 9-7 describes the condition under which the events are posted and severity level of the event.

Table 9-7 Credential Mapping Audit Events

Component	Description	Severity
Security Framework	The WebLogic Security Framework implements this interface and may post audit events for the following security events: Credentials for a WebLogic Server User are requested Credentials for a Subject are requested	Success

AuditLifecycleEvent

The AuditLifecycleEvent interface is used to post audit lifecycle events. The WebLogic Security Framework implements this interface and may post audit events for the following security events:

- After the auditing service in the framework is started.
- Before the auditing service in the framework is stopped.

The actual audit string returned by getEventType is `String eventType = "Event Type = AuditLifecycle Audit Event"`.

The `AuditLifecycleEventType` class describes the audit service lifecycle event types that are supported. Possible values are `START_AUDIT` and `STOP_AUDIT`.

An Auditing provider can use this interface to get additional information about the audit lifecycle event. The `AuditSeverity` and `AuditLifecycleEventType` attributes can be used to determine which of the above audit events has been posted.

AuditMgmtEvent

Management audit events are not currently posted by either the Security Framework or by the supplied providers. However, a custom security provider may implement this interface and post different audit events for the various management operations performed by the custom security provider.

An Auditing provider can use this interface to get additional information about the management audit event. The `AuditSeverity` attribute can be used to determine whether the management operation succeeded or failed.

AuditPolicyEvent

`AuditPolicyEvent` is posted by the security framework and the WebLogic Authorization provider. The security framework posts audit policy events when policies are deployed to or undeployed from an authorization provider. The WebLogic Server authorization provider posts audit policy events when creating, deleting, or updating policies. You can use the `getEventType` method to get the type of the audit event. The audit events and the actual audit strings returned by `getEventType` are as follows:

- `AuditStartPolicyDeployEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = Authorization Start Policy Deploy Audit Event ".`)
- `AuditPolicyUndeployEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = Authorization Policy Undeploy Audit Event ".`)
- `AuditPolicyDeployEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = Authorization Policy Deploy Audit Event ".`)
- `AuditPolicyDeleteAppEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = Authorization Delete Application Policies Audit Event ".`)
- `AuditEndPolicyDeployEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = Authorization End Policy Deploy Audit Event ".`)

For `PolicyConsumerAuditEvent`, which implements `AuditPolicyEvent`, the actual audit strings returned by `getEventType` are:

- `String eventType = "Event Type = Policy Consumer Get Handler"`
- `String eventType = "Event Type = Policy Consumer Set Policy"`
- `String eventType = "Event Type = Policy Consumer Set Unchecked Policy"`
- `String eventType = "Event Type = Policy Consumer Done"`

Table 9-8 describes the conditions under which the events are posted and lists the event severity level.

Table 9-8 Audit Policy Events

Component	Description	Severity
WebLogic Authorization Provider	• The WebLogic Authorization provider implements this interface and posts audit events for the following security events: • Security policy creation has succeeded. • Security policy creation has failed. • Security policy removal has succeeded. • Security policy removal has failed. • A security policy update has succeeded. • A security policy update has failed. • Application deletion of security policies has succeeded. • Application deletion of security policies has failed.	Success or Failure

AuditRoleDeploymentEvent

The security framework posts audit role deployment events when roles are deployed to or undeployed from a role mapping provider. You can use the `getEventType` method to get the type of the audit event. The following events are posted:

- `AuditRoleDeployEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = RoleManager Deploy Audit Event ".`)
- `AuditStartRoleDeployEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = RoleManager Start Deploy Role Audit Event ".`)
- `AuditEndRoleDeployEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = RoleManager End Deploy Role Audit Event ".`)
- `AuditRoleUndeployEvent` (The actual audit string returned by `getEventType` is `String eventType = "Event Type = RoleManager Undeploy Audit Event ".`)

Table 9-9 describes the conditions under which the events are posted and lists the event severity level.

Table 9-9 Audit Role Deployment Events

Component	Description	Severity
Security Framework	The WebLogic Security Framework implements this interface and may post audit events for the following security events: • Security role deployment to a role mapping provider has succeeded. • Security role deployment to a role mapping provider has failed. • Security role undeployment to a role mapping provider has succeeded. • Security role undeployment to a role mapping provider has failed. • Application deletion of security roles to a role mapping provider has succeeded. • Application deletion of security roles to a role mapping provider has failed.	Success or Failure

AuditRoleEvent

The WebLogic Authorization provider posts audit role events when roles are created, deleted, or updated. You can use the `getEventType` method to get the type of the audit event. The actual audit strings returned by `getEventType` are as follows:

- `String eventType = "Event Type = RoleManager Audit Event "`
- `String eventType = "Event Type = RoleManager Delete Application Roles Audit Event "`

For `RoleConsumerAuditEvent`, which implements `AuditRoleEvent`, the actual audit strings returned by `getEventType` are:

- `String eventType = "Event Type = Role Consumer Get Handler"`
- `String eventType = "Event Type = Role Consumer Set Role"`
- `String eventType = "Event Type = Role Consumer Done"`

Table 9-10 describes the conditions under which the events are posted and lists the event severity level.

Table 9-10 Audit Role Events

Component	Description	Severity
WebLogic Authorization Provider	The WebLogic Authorization provider implements this interface and posts audit events for the following security events: • Security role creation has succeeded. • Security role creation has failed. • Security role removal has succeeded. • Security role removal has failed. • A security role update has succeeded. • A security role update has failed.	Success

10

Credential Mapping Providers

This chapter describes credential mapping provider concepts and functionality, and provides step-by-step instructions for developing a custom credential mapping provider.
Credential mapping is the process whereby a legacy system's database is used to obtain an appropriate set of credentials to authenticate users to a target resource. In WebLogic Server, a credential mapping provider is used to provide credential mapping services and bring new types of credentials into the WebLogic Server environment.

This chapter includes the following sections:

Credential Mapping Concepts

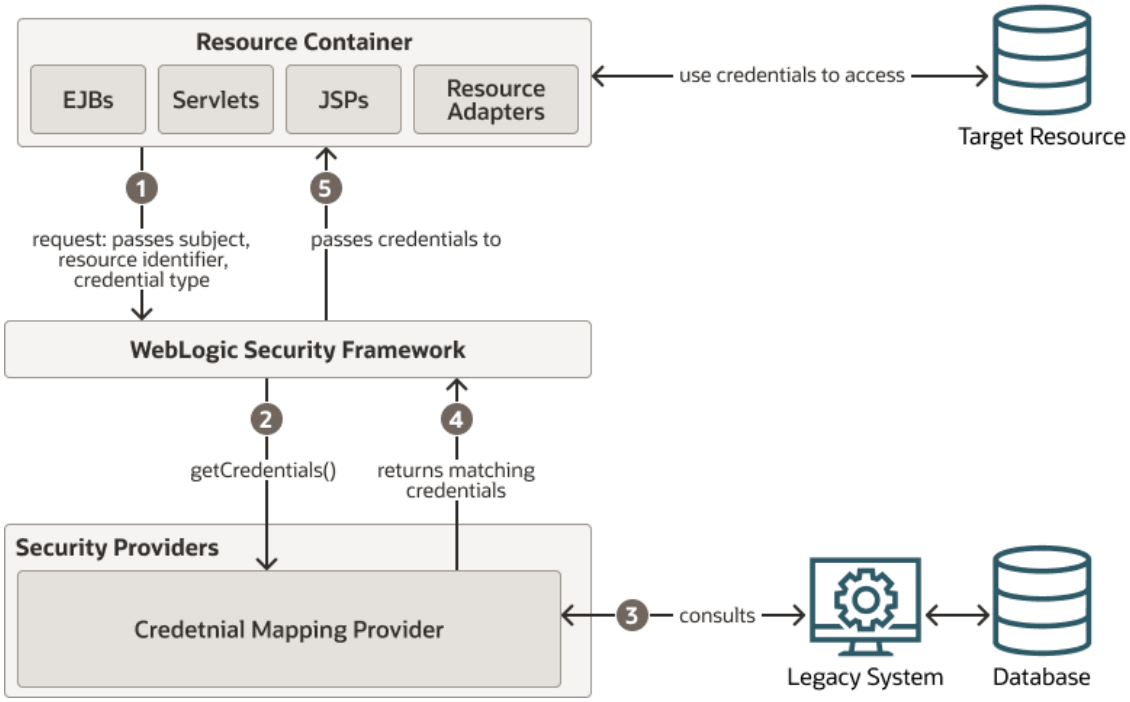
A **subject**, or source of a WebLogic resource request, has security-related attributes called **credentials**. A credential may contain information used to authenticate the subject to new services. Such credentials include username/password combinations, Kerberos tickets, and public key certificates. Credentials might also contain data that allows a subject to perform certain activities. Cryptographic keys, for example, represent credentials that enable the subject to sign or encrypt data.

A **credential map** is a mapping of credentials used by WebLogic Server to credentials used in a legacy (or any remote) system, which tell WebLogic Server how to connect to a given resource in that system. In other words, credential maps allow WebLogic Server to log in to a remote system on behalf of a subject that has already been authenticated. You can map credentials in this way by developing a credential mapping provider.

The Credential Mapping Process

Figure 10-1 illustrates how credential mapping providers interact with the WebLogic Security Framework during the credential mapping process, and an explanation follows.

Figure 10-1 Credential Mapping Providers and the Credential Mapping Process



Generally, credential mapping is performed in the following manner:

1. Application components, such as JavaServer Pages (JSPs), servlets, Enterprise JavaBeans (EJBs), or Resource Adapters call into the WebLogic Security Framework through the appropriate resource container. As part of the call, the application component passes in the subject (that is, the who making the request), the WebLogic resource (that is, the what that is being requested) and information about the type of credentials needed to access the WebLogic resource.
2. The WebLogic Security Framework sends the application component's request for credentials to a configured credential mapping provider. It is up to the credential mapper to decide whether it supports the token or not. If it supports the token, it performs its processing.
3. The credential mapping provider consults the legacy system's database to obtain a set of credentials that match those requested by the application component.
4. The credential mapping provider returns the credentials to the WebLogic Security Framework.
5. The WebLogic Security Framework passes the credentials back to the requesting application component through the resource container.

   The application component uses the credentials to access the external system. The external system might be a database resource, such as an Oracle or SQL Server.

Do You Need to Develop a Custom Credential Mapping Provider?

The default (that is, active) security realm for WebLogic Server includes a WebLogic Credential Mapping provider. The WebLogic Credential Mapping provider maps WebLogic Server users

and groups to the appropriate username/password credentials that may be required by other, external systems. If the type of credential mapping you want is between WebLogic Server users and groups and username/password credentials in another system, then the WebLogic Credential Mapping provider is sufficient.

WebLogic Server includes a PKI Credential Mapping provider. The PKI (Public Key Infrastructure) Credential Mapping provider included in WebLogic Server maps a WebLogic Server subject (the initiator) and target resource (and an optional credential action) to a key pair or public certificate that should be used by the application when using the targeted resource. The PKI Credential Mapping provider uses the subject and resource name to retrieve the corresponding credential from the keystore. The PKI Credential Mapping provider supports the `CredentialMapperV2.PKI_KEY_PAIR_TYPE` and `CredentialMapperV2.PKI_TRUSTED_CERTIFICATE_TYPE` token types.

WebLogic Server also includes the SAML 2.0 Credential Mapping provider. The SAML 2.0 Credential Mapping provider generates SAML 2.0 assertions for authenticated subjects based on a target site or resource. If the requested target has not been configured and no defaults are set, an assertion will not be generated. User information and group membership (if configured as such) are put in the AttributeStatement.

As described in Configuring SAML SSO Attribute Support in *Developing Applications with the WebLogic Security Service*, WebLogic Server enhanced the SAML 2.0 Credential Mapping provider and Identity Assertion provider mechanisms to support the use of a custom attribute mapper that can obtain additional attributes (other than group information) to be written into SAML assertions, and to then map attributes from incoming SAML assertions.

The SAML 2.0 Credential Mapping provider supports the `CredentialMapperV2.SAML2_ASSERTION_DOM_TYPE`, and `CredentialMapperV2.SAML2_ASSERTION_TYPE` token types.

If the out-of-the-box credential mapping providers do not meet your needs, then you need to develop a custom credential mapping provider. Note, however, that only the following token types are ever requested by the WebLogic Server resource containers:

- `CredentialMapperV2.PASSWORD_TYPE`
- `CredentialMapperV2.PKI_KEY_PAIR_TYPE`
- `CredentialMapperV2.PKI_TRUSTED_CERTIFICATE_TYPE`
- `CredentialMapperV2.SAML2_ASSERTION_DOM_TYPE`
- `CredentialMapperV2.SAML2_ASSERTION_TYPE`
- `CredentialMapperV2.USER_PASSWORD_TYPE`

Does Your Custom Credential Mapping Provider Need to Support Application Versioning?

All authorization, role mapping, and credential mapping providers for the security realm must support application versioning in order for an application to be deployed using versions. If you develop a custom security provider for authorization, role mapping, or credential mapping and need to support versioned applications, you must implement the Versionable Application SSPI, as described in Versionable Application Providers.

How to Develop a Custom Credential Mapping Provider

If the WebLogic Credential Mapping provider does not meet your needs, you can develop a custom credential mapping provider by following these steps:

1. Create Runtime Classes Using the Appropriate SSPIs
2. Generate an MBean type for your custom credential mapping provider by completing the steps described in Generate an MBean Type Using the WebLogic MBeanMaker.
3. Provide a Mechanism for Credential Map Management

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Determine Which Provider Interface You Will Implement
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom credential mapping provider by following these steps:

- Implement the CredentialProviderV2 SSPI *or* Implement the DeployableCredentialProvider SSPI
- Implement the CredentialMapperV2 SSPI

Implement the CredentialProviderV2 SSPI

To implement the `CredentialProviderV2` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following method:

- getCredentialProvider

  ```
  public CredentialMapperV2 getCredentialProvider();
  ```

  The `getCredentialProviderV2` method obtains the implementation of the `CredentialMapperV2` SSPI. For a single runtime class called `MyCredentialMapperProviderImpl.java` (as in Figure 2-3), the implementation of the `getCredentialProvider` method would be:

  ```
  return this;
  ```

  If there are two runtime classes, then the implementation of the `getCredentialProvider` method could be:

  ```
  return new MyCredentialMapperImpl;
  ```

  This is because the runtime class that implements the `CredentialProviderV2` SSPI is used as a factory to obtain classes that implement the `CredentialMapperV2` SSPI.

See *Java API Reference for Oracle WebLogic Server* for the CredentialProviderV2 SSPI.

Implement the DeployableCredentialProvider SSPI

> **Note**
>
> The DeployableCredentialProvider SSPI is deprecated in this release of WebLogic Server.

To implement the `DeployableCredentialProvider` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs, Implement the CredentialProviderV2 SSPI, *and* the following methods:

- deployCredentialMapping

  ```
  public void deployCredentialMapping(Resource resource, String
  initiatingPrincipal, String eisUsername, String eisPassword)throws
  ResourceCreationException;
  ```

  The `deployCredentialMapping` method deploys credential maps. If the mapping already exists, it is removed and replaced by this mapping. The `resource` parameter represents the WebLogic resource to which the initiating principal (represented as a `String`) is requesting access. The Enterprise Information System (EIS) username and password are the credentials in the legacy (remote) system to which the credential maps are being made.

- undeployCredentialMappings

  ```
  public void undeployCredentialMappings(Resource resource) throws
  ResourceRemovalException;
  ```

  The `undeployCredentialMappings` method undeploys credential maps (that is, deletes a credential mapping on behalf of an undeployed Resource Adapter from a database). The `resource` parameter represents the WebLogic resource for which the mapping should be removed.

> **Note**
>
> The deployCredentialMapping/undeployCredentialMappings methods operate on username/password credentials only.

See Java API Reference for Oracle WebLogic Server.

Implement the CredentialMapperV2 SSPI

The `CredentialMapperV2` interface defines the security service provider interface (SSPI) for objects capable of obtaining the appropriate set of credentials for a particular resource that is scoped within an application.

Only the following credential types are supported and passed to the `CredentialMapperV2` interface:

- PASSWORD_TYPE
- PKI_KEY_PAIR_TYPE

- PKI_TRUSTED_CERTIFICATE_TYPE
- SAML2_ASSERTION_DOM_TYPE
- SAML2_ASSERTION_TYPE
- USER_PASSWORD_TYPE

To implement the `CredentialMapperV2` SSPI, you must provide implementations for the following methods:

- getCredential

  ```
  public Object getCredential(Subject requestor, String initiator, Resource
  resource, ContextHandler handler, String credType);
  ```

  The `getCredential` method returns the credential of the specified type from the target resource associated with the specified initiator.

- getCredentials

  ```
  public Object[] getCredentials(Subject requestor, Subject initiator, Resource
  resource, ContextHandler handler, String credType);
  ```

  The `getCredentials` method returns the credentials of the specified type from the target resource associated with the specified initiator.

See *Java API Reference for Oracle WebLogic Server* for the CredentialMapperV2 SSPI.

Provide a Mechanism for Credential Map Management

While configuring a custom credential mapping provider using the WebLogic Remote Console makes it accessible by applications requiring credential mapping services, you also need to supply administrators with a way to manage this security provider's associated credential maps. The WebLogic Credential Mapping provider, for example, supplies administrators with a credential mappings page that allows them to add, modify, or remove credential mappings for various Connector modules.

Neither the credential mapping page nor access to it is available to administrators when you develop a custom credential mapping provider. Therefore, you must provide your own mechanism for credential map management. This mechanism must read and write credential maps to and from the custom credential mapping provider's database.

For more information, see Develop a Stand-Alone Tool for Credential Map Management.

Develop a Stand-Alone Tool for Credential Map Management

You would typically select this option if you want to develop a tool that is entirely separate from the WebLogic Remote Console.

For this option, you do not need to develop any management MBeans. However, your tool needs to:

1. Determine the WebLogic resource's ID. See WebLogic Resource Identifiers.
2. Determine how to represent the represent the local-to-remote user relationship. (This representation is entirely up to you and need not be a string.)
3. Read and write the expressions from and to the custom credential mapping provider's database.

11

Auditing Events From Custom Security Providers

This chapter describes the background information you need to understand before adding auditing capability to your custom security providers, and provides step-by-step instructions for adding auditing capability to a custom security provider.
As described in Auditing Providers **auditing** is the process whereby information about operating requests and the outcome of those requests are collected, stored, and distributed for the purposes of non-repudiation. Auditing providers provide this electronic trail of computer activity.

Each type of security provider can call the configured Auditing providers with a request to write out information about security-related events, before or after these events take place. For example, if a user attempts to access a `withdraw` method in a bank account application (to which they should not have access), the authorization provider can request that this operation be recorded. Security-related events are only recorded when they meet or exceed the severity level specified in the configuration of the Auditing providers.

This chapter includes the following sections:

- Security Services and the Auditor Service
- How to Audit From a Custom Security Provider

Security Services and the Auditor Service

The `SecurityServices` interface, located in the `weblogic.security.spi` package, is a repository for security services (currently just the Auditor Service). As such, the `SecurityServices` interface is responsible for supplying callers with a reference to the Auditor Service via the following method:

- getAuditorService

  ```
  public AuditorService getAuditorService
  ```

  The `getAuditorService` method returns the `AuditService` if an Auditing provider is configured.

The `AuditorService` interface, also located in the `weblogic.security.spi` package, provides other types of security providers (for example, authentication providers) with limited (write-only) auditing capabilities. In other words, the Auditor Service fans out invocations of each configured Auditing provider's `writeEvent` method, which simply writes an audit record based on the information specified in the `AuditEvent` object that is passed in.

See Implement the AuditChannel SSPIand Create an Audit Event. The `AuditorService` interface includes the following method:

- providerAuditWriteEvent

  ```
  public void providerAuditWriteEvent (AuditEvent event)
  ```

  The `providerAuditWriteEvent` method gives security providers *write access* to the object in the WebLogic Security Framework that calls the configured Auditing providers. The

`event` parameter is an `AuditEvent` object that contains the audit criteria, including the type of event to audit and the audit severity level. See Create an Audit Event and Audit Severity respectively.

The Auditor Service can be called to write audit events before or after those events have taken place, but does not maintain context in between pre and post operations. Security providers designed with auditing capabilities will need to obtain the Auditor Service as described in Obtain and Use the Auditor Service to Write Audit Events.

> **Note**
>
> Implementations for both the `SecurityServices` and `AuditorService` interfaces are created by the WebLogic Security Framework at boot time if an Auditing provider is configured. (See Configure the Custom Auditing Provider.) Therefore, you do not need to provide your own implementations of these interfaces.
>
> Additionally, `SecurityServices` objects are specific to the security realm in which your security providers are configured. Your custom security provider's runtime class automatically obtains a reference to the realm-specific `SecurityServices` object as part of its initialize method. (See Understand the Purpose of the Provider SSPIs.)

See *Java API Reference for Oracle WebLogic Server*for the SecurityServices and AuditorServiceinterfaces.

How to Audit From a Custom Security Provider

Add auditing capability to your custom security provider by following these steps:

- Create an Audit Event
- Obtain and Use the Auditor Service to Write Audit Events

Examples for each of these steps are provided in Example: Implementation of the AuditRoleEvent Interface and Example: Obtaining and Using the Auditor Service to Write Role Audit Events , respectively.

> **Note**
>
> If your custom security provider is to record audit events, be sure to include any classes created as a result of these steps into the MBean JAR File (MJF) for the custom security provider (that is, in addition to the other files that are required).

Create an Audit Event

Security providers must provide information about the events they want audited, such as the type of event (for example, an authentication event) and the audit severity (for example, error). **Audit Events** contain this information, and can also contain any other contextual data that is understandable to a configured Auditing provider. To create an Audit Event, either:

- Implement the AuditEvent SSPI or
- Implement an Audit Event Convenience Interface

Implement the AuditEvent SSPI

To implement the `AuditEvent` SSPI, provide implementations for the following methods:

- getEventType

  ```
  public java.lang.String getEventType()
  ```

  The `getEventType` method returns a string representation of the event type that is to be audited, which is used by the Audit Channel (that is, the runtime class that implements the `AuditChannel` SSPI). For example, the event type for the Oracle-provided implementation is `Authentication Audit Event`. See Audit Channels and Implement the AuditChannel SSPI.

- getFailureException

  ```
  public java.lang.Exception getFailureException()
  ```

  The `getFailureException` method returns an `Exception` object, which is used by the Audit Channel to obtain audit information, in addition to the information provided by the `tostring` method.

- getSeverity

  ```
  public AuditSeverity getSeverity()
  ```

  The `getSeverity` method returns the severity level value associated with the event type that is to be audited, which is used by the Audit Channel. This allows the Audit Channel to make the decision about whether or not to audit. See Audit Severity.

- toString

  ```
  public java.lang.String toString()
  ```

  The `toString` method returns preformatted audit information to the Audit Channel.

> **Note**
>
> The `toString` method can produce any character and no escaping is used. If your Audit provider is writing the `toString` value into a format that uses characters for syntax, escape the `toString` value before writing it.

See *Java API Reference for Oracle WebLogic Server* for the AuditEvent SSPI.

Implement an Audit Event Convenience Interface

There are several subinterfaces of the `AuditEvent` SSPI that are provided for your convenience, and that can assist you in structuring and creating Audit Events.

Each of these Audit Event convenience interfaces can be used by an Audit Channel (that is, a runtime class that implements the `AuditChannel` SSPI) to more effectively determine the instance types of extended event type objects, for a certain type of security provider. For example, the `AuditAtnEventV2` convenience interface can be used by an Audit Channel that wants to determine the instance types of extended authentication event type objects. (See Audit Channels and Implement the AuditChannel SSPI.)

The Audit Event convenience interfaces are:

- The AuditAtnEventV2 Interface
- The AuditAtzEvent and AuditPolicyEvent Interfaces
- The AuditMgmtEvent Interface
- The AuditRoleEvent and AuditRoleDeploymentEvent Interfaces

> **Note**
>
> It is recommended, but not required, that you implement one of the Audit Event convenience interfaces.

The AuditAtnEventV2 Interface

The `AuditAtnEventV2` convenience interface helps Audit Channels to determine instance types of extended authentication event type objects.

> **Note**
>
> The `AuditAtnEvent` interface is deprecated in this release of WebLogic Server.

To implement the `AuditAtnEventV2` interface, provide implementations for the methods described in Implement the AuditEvent SSPI *and* the following methods:

- getUsername

  ```
  public String getUsername()
  ```

  The `getUsername` method returns the username associated with the authentication event.

- getAtnEventType

  ```
  public AuditAtnEventV2.AtnEventTypeV2 getAtnEventType()
  ```

  The `getAtnEventType` method returns an event type that more specifically represents the authentication event. The specific authentication event types are:

  `AUTHENTICATE`: simple authentication using a username and password occurred.

  `ASSERTIDENTITY`: perimeter authentication based on tokens occurred.

  `CREATEDERIVEDKEY`: represents the creation of the Derived key.

  `CREATEPASSWORDDIGEST`: represents the creation of the Password Digest.

  `IMPERSONATEIDENTITY`: client identity has been established using the supplied client username (requires kernel identity).

  `USERLOCKED`: a user account has been locked because of invalid login attempts.

  `USERUNLOCKED`: a lock on a user account has been cleared.

  `USERLOCKOUTEXPIRED`: a lock on a user account has expired.

  `VALIDATEIDENTITY`: authenticity (trust) of the principals within the supplied subject has been validated.

- toString

```
public String toString()
```

The `toString` method returns the specific authentication information to audit, represented as a string.

> **Note**
>
> The `toString` method can produce any character and no escaping is used. If your Audit provider is writing the `toString` value into a format that uses characters for syntax, escape the `toString` value before writing it.
>
> The `AuditAtnEventV2` convenience interface extends both the `AuditEvent` and `AuditContext` interfaces. For more information about the AuditContext interface, see Audit Context.

See *Java API Reference for Oracle WebLogic Server* for the AuditAtnEventV2 interface.

The AuditAtzEvent and AuditPolicyEvent Interfaces

The `AuditAtzEvent` and `AuditPolicyEvent` convenience interfaces help Audit Channels to determine instance types of extended authorization event type objects.

> **Note**
>
> The difference between the `AuditAtzEvent` convenience interface and the `AuditPolicyEvent` convenience interface is that the latter only extends the `AuditEvent` interface. (It does not also extend the `AuditContext` interface.) See Audit Context.

To implement the `AuditAtzEvent` or `AuditPolicyEvent` interface, provide implementations for the methods described in Implement the AuditEvent SSPI *and* the following methods:

- getSubject

  ```
  public Subject getSubject()
  ```

  The `getSubject` method returns the subject associated with the authorization event (that is, the subject attempting to access the WebLogic resource).

- getResource

  ```
  public Resource getResource()
  ```

  The `getResource` method returns the WebLogic resource associated with the authorization event that the subject is attempting to access.

See *Java API Reference for Oracle WebLogic Server* for the AuditAtzEvent and AuditPolicyEvent interfaces.

The AuditMgmtEvent Interface

The `AuditMgmtEvent` convenience interface helps Audit Channels to determine instance types of extended security management event type objects, such as a security provider's MBean. It contains no methods that you must implement, but maintains the best practice structure for an Audit Event implementation.

> **Note**
>
> See Security Service Provider Interface (SSPI) MBeans.

See *Java API Reference for Oracle WebLogic Server* for the AuditMgmtEvent interface.

The AuditRoleEvent and AuditRoleDeploymentEvent Interfaces

The `AuditRoleDeploymentEvent` and `AuditRoleEvent` convenience interfaces help Audit Channels to determine instance types of extended role mapping event type objects. They contain no methods that you must implement, but maintain the best practice structure for an Audit Event implementation.

> **Note**
>
> The difference between the `AuditRoleEvent` convenience interface and the `AuditRoleDeploymentEvent` convenience interface is that the latter only extends the `AuditEvent` interface. (It does not also extend the `AuditContext` interface.) See Audit Context.

See *Java API Reference for Oracle WebLogic Server* for the AuditRoleEvent and AuditRoleDeploymentEventinterfaces.

Audit Severity

The **audit severity** is the level at which a security provider wants audit events to be recorded. When the configured Auditing providers receive a request to audit, each will examine the severity level of events taking place. If the severity level of an event is greater than or equal to the level an Auditing provider was configured with, that Auditing provider will record the audit data.

> **Note**
>
> Auditing providers are configured using the WebLogic Remote Console. See Configure the Custom Auditing Provider.

The `AuditSeverity` class, which is part of the `weblogic.security.spi` package, provides audit severity levels as both numeric and text values to the Audit Channel (that is, the `AuditChannel` SSPI implementation) through the `AuditEvent` object. The numeric severity value is to be used in logic, and the text severity value is to be used in the composition of the audit record output. See Implement the AuditChannel SSPI and Create an Audit Eventrespectively.

Audit Context

Some of the Audit Event convenience interfaces extend the `AuditContext` interface to indicate that an implementation will also contain contextual information. This contextual information can then be used by Audit Channels. See Audit Channels and Implement the AuditChannel SSPI.

The `AuditContext` interface includes the following method:

- getContext

  ```
  public ContextHandler getContext()
  ```

  The `getContext` method returns a `ContextHandler` object, which is used by the runtime class (that is, the `AuditChannel` SSPI implementation) to obtain additional audit information. See ContextHandlers and WebLogic Resources.

Example: Implementation of the AuditRoleEvent Interface

Example 11-1 shows the `MyAuditRoleEventImpl.java` class, which is a sample implementation of an Audit Event convenience interface (in this case, the `AuditRoleEvent` convenience interface). This class includes implementations for:

- The four methods inherited from the `AuditEvent` SSPI: `getEventType`, `getFailureException`, `getSeverity` and `toString` (as described in Implement the AuditEvent SSPI).
- One additional method: `getContext`, which returns additional contextual information via the ContextHandler. (See ContextHandlers and WebLogic Resources.)

> **Note**
>
> The bold face code in Example 11-1 highlights the class declaration and the method signatures.

Example 11-1 MyAuditRoleEventImpl.java

```
package mypackage;
import jakarta.security.auth.Subject;
import weblogic.security.SubjectUtils;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AuditRoleEvent;
import weblogic.security.spi.AuditSeverity;
import weblogic.security.spi.Resource;
/*package*/ class MyAuditRoleEventImpl implements AuditRoleEvent
{
   private Subject subject;
   private Resource resource;
   private ContextHandler context;
   private String details;
   private Exception failureException;
   /*package*/ MyAuditRoleEventImpl(Subject subject, Resource resource,
      ContextHandler context, String details, Exception
      failureException) {
         this.subject = subject;
         this.resource = resource;
         this.context = context;
         this.details = details;
         this.failureException = failureException;
   }
   public Exception getFailureException()
   {
      return failureException;
   }
   public AuditSeverity getSeverity()
   {
```

```
      return (failureException == null) ? AuditSeverity.SUCCESS :
         AuditSeverity.FAILURE;
   }
   public String getEventType()
   {
      return "MyAuditRoleEventType";
   }
   public ContextHandler getContext()
   {
      return context;
   }
   public String toString()
   {
      StringBuffer buf = new StringBuffer();
      buf.append("EventType:" + getEventType() + "\n");
      buf.append("\tSeverity: " +
         getSeverity().getSeverityString());
      buf.append("\tSubject: " +
         SubjectUtils.displaySubject(getSubject());
      buf.append("\tResource: " + resource.toString());
      buf.append("\tDetails: " + details);
      if (getFailureException() != null) {
         buf.append("\n\tFailureException:" +
            getFailureException());
      }
      return buf.toString();
   }
}
```

Obtain and Use the Auditor Service to Write Audit Events

To obtain and use the Auditor Service to write audit events from a custom security provider, follow these steps:

1. Use the `getAuditorService` method to return the Audit Service.

   > **Note**
   >
   > Recall that a `SecurityServices` object is passed into a security provider's implementation of a Provider SSPI as part of the initialize method. (For more information, see Understand the Purpose of the Provider SSPIs.) An `AuditorService` object will only be returned if an Auditing provider has been configured.

2. Instantiate the Audit Event you created in Implement the AuditEvent SSPI and send it to the Auditor Service through the `AuditService.providerAuditWriteEvent` method.

Example: Obtaining and Using the Auditor Service to Write Role Audit Events

Example 11-2 illustrates how a custom role mapping provider's runtime class (called `MyRoleMapperProviderImpl.java`) would obtain the Auditor Service and use it to write out audit events.

> **Note**
>
> The `MyRoleMapperProviderImpl.java` class relies on the `MyAuditRoleEventImpl.java` class from Example 11-1.

Example 11-2 MyRoleMapperProviderImpl.java

```
package mypackage;
import jakarta.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.SubjectUtils;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AuditorService;
import weblogic.security.spi.RoleMapper;
import weblogic.security.spi.RoleProvider;
import weblogic.security.spi.Resource;
import weblogic.security.spi.SecurityServices;
public final class MyRoleMapperProviderImpl implements RoleProvider, RoleMapper
{
   private AuditorService auditor;
   public void initialize(ProviderMBean mbean, SecurityServices
      services)
   {
      auditor = services.getAuditorService();
      ...
   }
   public Map getRoles(Subject subject, Resource resource,
      ContextHandler handler)
   {
      ...
      if (auditor != null)
      {
         auditor.providerAuditWriteEvent(
            new MyRoleEventImpl(subject, resource, context,
            "why logging this event",
            null);               // no exception occurred
      }
      ...
   }
}
```

Auditing Management Operations from a Provider's MBean

A `SecurityServices` object is passed into a security provider's implementation of a "Provider" SSPI as part of the `initialize` method. (See Understand the Purpose of the Provider SSPIs.) The provider can use this object's auditor to audit provider-specific security events, such as when a user is successfully logged in.

A security provider's MBean implementation is not passed a SecurityServices object. However, the provider may need to audit its MBean operations, such as a user being created.

To work around this, the provider's runtime implementation can cache the SecurityServices object and use a provider-specific mechanism to pass it to the provider's MBean implementation. This allows the provider to audit its MBean operations.

The Manageable Sample Authentication Provider shows one way to accomplish this task. The sample provider contains three major implementation classes:

- ManageableSampleAuthenticationProviderImpl contains its security runtime implementation.
- ManageableSampleAuthenticatorImpl contains its MBean implementation.
- UserGroupDatabase is a helper class used by ManageableSampleAuthenticationProviderImpl and ManageableSampleAuthenticatorImpl.

The code flow to cache and obtain the SecurityServices object is as follows:

1. The ManageableSampleAuthenticationProviderImpl's `initialize` method is passed a SecurityServices object.
2. The `initialize` method creates a UserGroupDataBase object and passes it the SecurityServices object.
3. The UserGroupDataBaseObject caches the SecurityServices object. The `initialize` method also puts the UserGroupDatabase object into a hash table using the realm's name as the lookup key.
4. The ManageableSampleAuhenticatorImpl's `init` method finds its realm name from its MBean.
5. The `init` method uses the realm name to find the corresponding UserGroupDataBase object from the hash table.
6. The `init` method then retrieves the SecurityServices object from the UserGroupDatabase object, and uses its auditor to audit management operations such as "createUser."

> **Note**
>
> A provider's runtime implementation is initialized only if the provider is part of the default realm when the server is booted. Therefore, if the provider is not in the default realm when the server is booted, its runtime implementation is never initialized, and the provider's MBean implementation cannot gain access to the `SecurityServices` object. That is, if the provider is not in the default realm when the server is booted, the provider cannot audit its MBean operations.

Example: Auditing Management Operations from a Provider's MBean

Example 11-3 illustrates how the ManageableSampleAuhenticatorImpl's `init` method finds its realm name from its MBean, how it uses the realm name to find the corresponding `UserGroupDataBase` object from the hash table (via the `UserGroupDatabase` helper class), and how it then retrieves the `SecurityServices` object from the `UserGroupDatabase` object.

Example 11-3 also shows how `ManageableSampleAuhenticatorImpl` uses its auditor to audit management operations such as "createUser."

Example 11-3 ManageableSampleAuthenticatorImpl.java

```
package examples.security.providers.authentication.manageable;
import java.util.Enumeration;
import javax.management.MBeanException;
import javax.management.modelmbean.ModelMBean;
import weblogic.management.security.authentication.AuthenticatorImpl;
import weblogic.management.utils.AlreadyExistsException;
import weblogic.management.utils.InvalidCursorException;
import weblogic.management.utils.NotFoundException;
import weblogic.security.spi.AuditorService;
import weblogic.security.spi.SecurityServices;
```

```
public class ManageableSampleAuthenticatorImpl extends AuthenticatorImpl
{
// Manages the user and group definitions for this provider:
private UserGroupDatabase database;

// Manages active queries (see listUsers, listGroups, listMemberGroups):
private ListManager listManager = new ListManager();

// The name of the realm containing this provider:
private String realm;

// The name of this provider:
private String provider;

// The auditor for auditing user/group management operations.
// This is only available if this provider was configured in
// the default realm when the server was booted.
private AuditorService auditor;

public ManageableSampleAuthenticatorImpl(ModelMBean base) throws MBeanException
{
super(base);
}

private synchronized void init() throws MBeanException
{
if (database == null) {
try {
ManageableSampleAuthenticatorMBean myMBean = (ManageableSampleAuthenticatorMBean)getProxy();
database = UserGroupDatabase.getDatabase(myMBean);
realm    = myMBean.getRealm().getName();
provider = myMBean.getName();
SecurityServices services = database.getSecurityServices();
auditor = (services != null) ? services.getAuditorService() : null;
}
catch(Exception e) {
throw new MBeanException(e, "SampleAuthenticatorImpl.init failed");
}
}
}
...
public void createUser(String user, String password, String description)
throws MBeanException, AlreadyExistsException
{
init();
String details = (auditor != null) ?
"createUser(user = " + user + ", password = " + password + ",
description = " + description + ")" : null;
try {
// we don't support descriptions so just ignore it
database.checkDoesntExist(user);
database.getUser(user).create(password);
database.updatePersistentState();
auditOperationSucceeded(details);
}
catch (AlreadyExistsException   e) { auditOperationFailed(details, e); throw e; }
catch (IllegalArgumentException e) { auditOperationFailed(details, e); throw e; }
}
...
private void auditOperationSucceeded(String details)
{
```

```
if (auditor != null) {
auditor.providerAuditWriteEvent(
new ManageableSampleAuthenticatorManagementEvent(realm, provider, details, null)
);
}
}
...
private void auditOperationFailed(String details, Exception failureException)
{
if (auditor != null) {

auditor.providerAuditWriteEvent(
new ManageableSampleAuthenticatorManagementEvent(realm, provider, details, failureException)
);
}
}
}
```

Best Practice: Posting Audit Events from a Provider's MBean

Provider's management operations that do writes (for example, create user, delete user, remove data) should post audit events, regardless of whether or not the operation succeeds.

If your provider audits MBean operations, you should keep the following Best Practice guidelines in mind.

- If the write operation succeeds, post an INFORMATION audit event.
- If the write operation fails because of a bad parameter (for example, because the user already exists, or due to a bad import format name, a non-existent file name, or the wrong file format), do not post an audit event.
- If the write operation fails because of an error (for example, LDAPException, RuntimeException), post a FAILURE audit event.
- Import operations can partially succeed. For example, some of the users are imported, but others are skipped because there are already users with that name in the provider.
- If you can easily detect that the data you are skipping is identical to the data already in the provider (for example, the username, description, and password are the same) then consider posting a WARNING event.
- If you are skipping data because there is a partial collision (for example, the username is the same but the password is different), you should post a FAILURE event.
- If it is too difficult to distinguish the import data from the data already stored in the provider, post a FAILURE event.

12

Servlet Authentication Filters

This chapter describes Servlet Authentication Filter interface concepts and functionality, and provides step-by-step instructions for developing a Servlet Authentication Filter.
A Servlet Authentication Filter is a provider type that performs pre- and post-processing for authentication functions, including identity assertion. A Servlet Authentication Filter is a special type of security provider that primarily acts as a helper to an authentication provider.

The `ServletAuthenticationFilter` interface defines the security service provider interface (SSPI) for authentication filters that can be plugged in to WebLogic Server. You implement the `ServletAuthenticationFilter` interface as part of an authentication provider, and typically as part of the identity assertion form of authentication provider, to signal that the authentication provider has authentication filters that it wants the servlet container to invoke during the authentication process.

This chapter includes the following sections:

Authentication Filter Concepts

Filters, as defined by the Java Servlet API 2.3 specification, are preprocessors of the request before it reaches the servlet, and/or postprocessors of the response leaving the servlet. Filters provide the ability to encapsulate recurring tasks in reusable units and can be used to transform the response from a servlet or JSP page.

Servlet Authentication filters are an extension to of the filter object that allows filters to replace or extend container-based authentication.

Why Filters are Needed

The WebLogic Security Framework allows you to provide a custom authentication provider. However, due to the nature of the Java Servlet API 2.3 specification, the interaction between the authentication provider and the client or other servers is architecturally limited during the authentication process. This restricts authentication mechanisms to those that are compatible with the authentication mechanisms the Servlet container offers: basic, form, and certificate.

Filters have fewer architecturally-dependence limitations; that is, they are not dependent on the authentication mechanisms offered by the Servlet container. By allowing filters to be invoked prior to the container beginning the authentication process, a security realm can implement a wider scope of authentication mechanisms. For example, a Servlet Authentication Filter could redirect the user to a SAML provider site for authentication.

JAAS LoginModules (within a WebLogic Authentication provider) can be used for customization of the login process. Customizing the location of the user database, the types of proof material required to execute a login, or the population of the Subject with groups is implemented via a LoginModule.

Conversely, redirecting to a remote site to execute the login, extracting login information out of the query string, and negotiating a login mechanism with a browser are implemented via a Servlet Authentication Filter.

Servlet Authentication Filter Design Considerations

You should consider the following design considerations when writing Servlet Authentication Filters:

- Do you need to allow multiple filters to be specified? You might want to allow this so that administrative decisions can be made at configuration time.
- Do you depend on a particular order of-execution? Servlet Authentication Filters must not be dependent on the order in which filters are executed.
- Have you considered allowing each filter to process the request both before and after authentication? If so, the filter should not make any assumptions about when it is being invoked.
- Consider allowing each filter to have the option of stopping the execution of the remaining filters and the Servlet's authentication process by not calling the Filter doFilter method.
- Do you need to allow a filter to cause the browser to redirect?
- Consider allowing a filter to work for 1-way SSL, 2-way SSL, identity assertion, form authentication, and basic authentication. For example, Form authentication is a two-request process and the filter is called twice for form authentication.

How Filters Are Invoked

The Servlet Authentication Filter interface allows an authentication provider to implement zero or more Servlet Authentication Filter classes. The filters are invoked as follows:

1. The servlet container calls the Servlet Authentication Filters prior to authentication occurring.

   The servlet container gets the configured chain of Servlet Authentication Filters from the WebLogic Security Framework.

   The Security Framework returns the Servlet Authentication Filters in the order of the authentication providers. If one provider has multiple Servlet Authentication Filters, the Security Framework uses the ordered list of jakarta.servlet.Filters returned by the ServletAuthenticationFilter `getAuthenticationFilters` method.

   Duplicate filters are allowed because they might need to execute multiple times to correctly manipulate the request.
2. For each filter, the servlet container calls the Filter `init` method to indicate to a filter that it is being placed into service.
3. The servlet container calls the Filter `doFilter` method on the first filter each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain.

   The `FilterChain` object passed in to this method allows the Filter to pass on the request and response to the next entity in the chain. Filters use the `FilterChain` object to invoke the next filter in the chain, or if the calling filter is the last filter in the chain, to invoke the resource at the end of the chain.

4. If all Servlet Authentication Filters call the Filter `doFilter` method then, when the final one calls the `doFilter` method, the servlet container then performs authentication as it would if the filters were not present.

   However, if any of the Servlet Authentication Filters do not call the `doFilter` method, the remaining filters, the servlet, and the servlet container's authentication procedure are not called. This allows a filter to replace the servlet's authentication process. This typically involves authentication failure or redirecting to another URL for authentication.

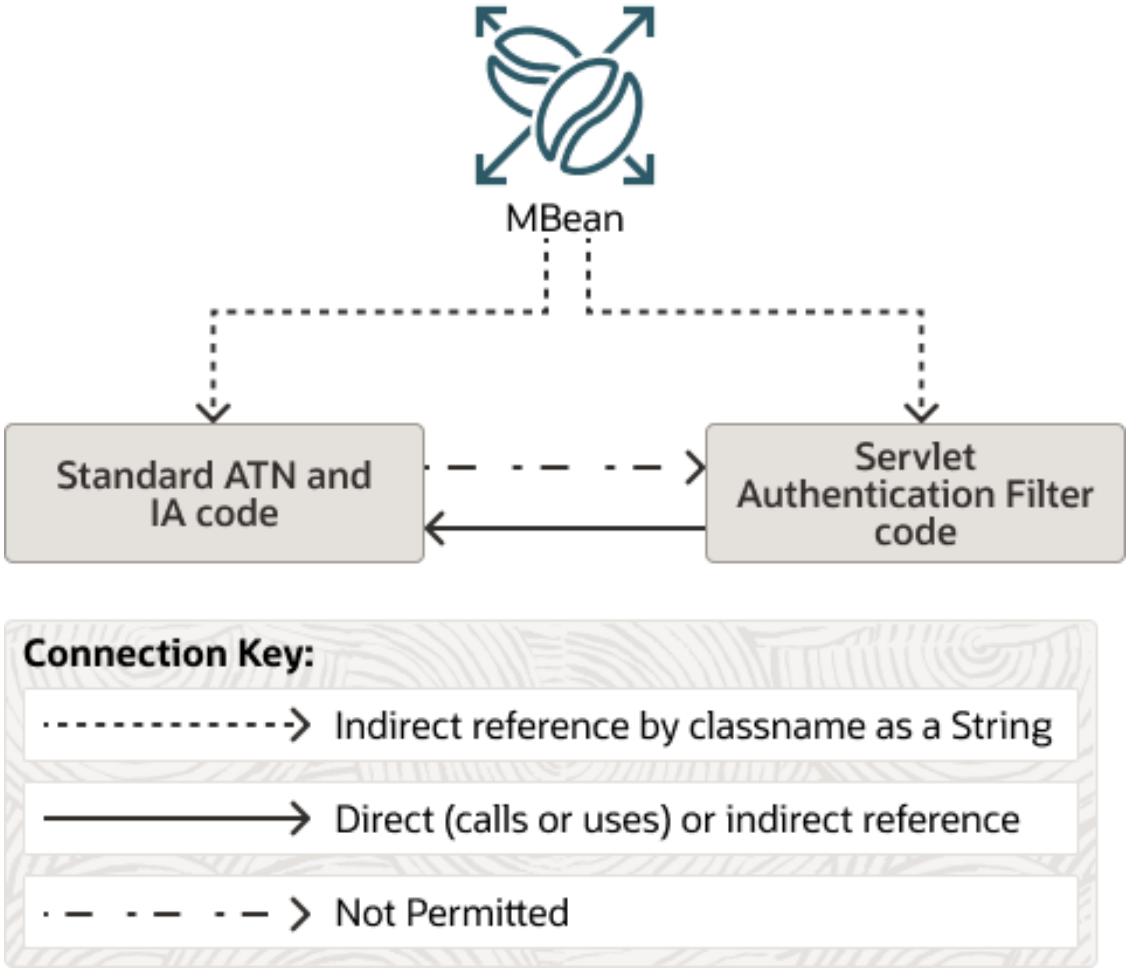
Do Not Call Servlet Authentication Filters From Authentication Providers

Although you implement the Servlet Authentication Filter interface as part of an authentication provider, authentication providers do not actually call Servlet Authentication Filters directly. The implementation of Servlet Authentication Filters depends upon particular features of the WebLogic Security Framework that know how to locate and invoke the filters.

If you develop a custom Servlet Authentication Filter, make sure that your custom authentication providers do not call the WLS-specific classes (for example, `weblogic.servlet.*`) and the Jakarta EE-specific classes (for example, `jakarta.servlet.*`). Following this rule ensures maximum portability with WebLogic Security.

Figure 12-1 illustrates this requirement.

Figure 12-1 Authentication Providers Do Not Call Servlet Authentication Filters



Example of a Provider that Implements a Filter

WebLogic Server includes a Servlet Authentication Filter that handles the header manipulation required by the Simple and Protected Negotiate (SPNEGO). This Servlet Authentication Filter, called the Negotiate Servlet Authentication Filter, is configured to support the WWW-Authenticate and Authorization HTTP headers.

The Negotiate Servlet Authentication Filter generates the appropriate WWW-Authenticate header on unauthorized responses for the negotiate protocol and handles the authorization headers on subsequent requests. The filter is available through the Negotiate Identity Assertion Provider.

By default, the Negotiate Identity Assertion provider is available, but not configured, in the WebLogic default security realm. The Negotiate Identity Assertion provider can be used instead of, or in addition to, the WebLogic Identity Assertion provider.

How to Develop a Custom Servlet Authentication Filter

You can develop a custom Servlet Authentication Filter by following these steps:

1. Create Runtime Classes Using the Appropriate SSPIs
2. Generate an MBean Type Using the WebLogic MBeanMaker
3. Configure the Authentication Provider

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your Servlet Authentication Filter by following these steps:

- Implement the AuthenticationProviderV2 SSPI *or* Implement the IdentityAsserterV2 SSPI
- Implement the Servlet Authentication Filter SSPI
- Implement the Filter Interface Methods

For an example of how to create a runtime class for a custom Servlet Authentication Filter provider, see Generate an MBean Type Using the WebLogic MBeanMaker.

Implement the Servlet Authentication Filter SSPI

You implement the `ServletAuthenticationFilter` interface as part of an authentication provider to signal that the authentication provider has authentication filters that it wants the servlet container to invoke during the authentication process.

To implement the Servlet Authentication Filter SSPI, provide an implementation for the following method:

- get Servlet Authentication Filters

  ```
  public Filter[] getServletAuthenticationFilters
  ```

The `getServletAuthenticationFilters` method returns an ordered list of the jakarta.servlet.Filters that are executed during the authentication process of the Servlet container. The container may call this method multiple times to get multiple instances of the Servlet Authentication Filter. On each call, this method should return a list of new instances of the filters.

Implement the Filter Interface Methods

To implement the Filter interface methods, provide implementations for the following methods. In typical use, you would call `init()` once, doFilter() possibly many times, and destroy() once.

- destroy

  ```
  public void destroy()
  ```

  The `destroy` method is called by the web container to indicate to a filter that it is being taken out of service. This method is only called once all threads within the filter's doFilter method have exited, or after a timeout period has passed. After the web container calls this method, it does not call the `doFilter` method again on this instance of the filter.

  This method gives the filter an opportunity to clean up any resources that are being held (for example, memory, file handles, threads) and make sure that any persistent state is synchronized with the filter's current state in memory

- doFilter

  ```
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain
  chain)
  ```

  The `doFilter` method of the Filter is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain. The `FilterChain` passed in to this method allows the Filter to pass on the request and response to the next entity in the chain.

  A typical implementation of this method would follow the following pattern:

  1. Examine the request.

  2. Optionally, wrap the request object with a custom implementation to filter content or headers for input filtering.

  3. Optionally, wrap the response object with a custom implementation to filter content or headers for output filtering.

  4. Either invoke the next entity in the chain using the `FilterChain` object (`chain.doFilter()`), or do not pass on the request/response pair to the next entity in the filter chain to block the request processing.

  5. Directly set headers on the response after invocation of the next entity in the filter chain.

- init

  ```
  public void init(FilterConfig filterConfig)
  ```

  The `init` method is called by the web container to indicate to a filter that it is being placed into service. The servlet container calls the `init` method exactly once after instantiating the filter. The `init` method must complete successfully before the filter is asked to do any filtering work.

Implementing Challenge Identity Assertion from a Filter

As described in Identity Assertion Providers the Challenge Identity Assertion interface supports challenge response schemes in which multiple challenges, responses messages, and state are required. The Challenge Identity Asserter interface allows identity assertion providers to support authentication protocols such as Microsoft's Windows NT Challenge/Response (NTLM), Simple and Protected GSS-API Negotiation Mechanism (SPNEGO), and other challenge/response authentication mechanisms.

Servlet Authentication Filters allow you to implement a challenge/response protocol without being limited to the authentication mechanisms compatible with the Servlet container. However, because Servlet Authentication Filters operate outside of the authentication environment provided by the Security Framework, they cannot depend on the Security Framework to determine provider context, and require an API to drive the multiple-challenge identity assertion process.

The `weblogic.security.services.Authentication` class has been extended to allow multiple challenge/response identity assertion from a Servlet Authentication Filter. The methods and interface provide a wrapper for the `ChallengeIdentityAsserterV2` and `ProviderChallengeContext` SSPI interfaces so that you can invoke them from a Servlet Authentication Filter.

There is no other documented way to perform a multiple challenge/response dialog from a Servlet Authentication Filter within the context of the Security Framework. Your Servlet Authentication Filter cannot directly invoke the `ChallengeIdentityAsserterV2` and `ProviderChallengeContext` interfaces.

Therefore, if you plan to implement multiple challenge/response identity assertion from a filter, you need to implement the `ChallengeIdentityAsserterV2` and `ProviderChallengeContext` interfaces, and then use the `weblogic.security.services.Authentication` methods and `AppChallengeContect` interface to invoke them from a Servlet Authentication Filter.

The steps to accomplish this process are described in Identity Assertion Providers and are summarized here:

- Implement the AuthenticationProviderV2 SSPI or Implement the IdentityAsserterV2 SSPI
- Implement the ChallengeIdentityAsserterV2 Interface
- Implement the ProviderChallengeContext Interface
- Invoke the weblogic.security.services Challenge Identity Methods
- Invoke the weblogic.security.services AppChallengeContext Methods

Generate an MBean Type Using the WebLogic MBeanMaker

When you generate the MBean type for your custom authentication provider as described in Authentication Providers you must also implement the MBean for your Servlet Authentication Filter.

The `ServletAuthenticationFilter` MBean extends the `AuthenticationProvider` MBean. The `ServletAuthenticationFilter` MBean is a marker interface and has no methods.

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">
<MBeanType

Name          = "ServletAuthenticationFilter"
```

```
Package       = "weblogic.management.security.authentication"
Extends       = "weblogic.management.security.authentication.AuthenticationProvider"
PersistPolicy = "OnUpdate"
Abstract      = "true"
Description   = "The SSPI MBean that all Servlet Authentication   Filter providers must
extend.
This MBean is just a marker interface.  It has no methods on it."
>
</MBeanType>
```

Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF)

Once your have run your MDF through the WebLogic MBeanMaker to generate your intermediate files, and you have edited the MBean implementation file to supply implementations for the appropriate methods within it, you need to package the MBean files and the runtime classes for the custom authentication provider, including the Servlet Authentication Filter, into an MBean JAR File (MJF).

These steps are described in Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF).

Configure the Authentication Provider

Configuring a custom authentication provider that implements a Servlet Authentication Filter means that you are adding the custom authorization provider to your security realm, where it can be accessed by applications requiring authorization services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers.

The steps for configuring a custom authorization provider are described in Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

13

Versionable Application Providers

This chapter describes the background information you need to understand before adding application versioning capability to your custom security providers, and provides step-by-step instructions for adding application versioning capability to a custom security provider.
A versionable application is an application that has an application archive version specified in the manifest of the application archive (EAR file). Versionable applications can be deployed side-by-side and active simultaneously. Versionable applications allow multiple versions of an application, where security constraints can vary between the application versions.

The versionable application provider SSPI enables all security providers that support application versioning to be notified when versions are created and deleted. It also enables all security providers that support application versioning to be notified when non-versioned applications are removed.

This chapter includes the following sections:

Versionable Application Concepts

Redeployment of versionable applications is always done via side-by-side versions, unless the same archive version is specified in the subsequent redeployments. However, a versionable application has to be written in such a way that multiple versions of it can be run side-by-side without conflicts; that is, it does not make any assumption of the uniqueness of the application name, and so forth. For example, in the case where an applications may use the application name as a unique key for global data structures, such as database tables or LDAP stores, the applications would need to change to use the application identifier instead.

Production Redeployment is allowed only if the configured security providers support the application versioning security SSPI. All authorization, role mapping, and credential mapping providers for the security realm must support application versioning for an application to be deployed using versions.

See Developing Applications for Production Redeployment in *Developing Applications for Oracle WebLogic Server* for detailed information on how an application assigns an application version.

The Versionable Application Process

For a security provider to support application versioning, it must implement the versionable application SSPI. The WebLogic Security Framework calls the versionable application provider SSPI when an application version is created and deleted so that the provider can take any required actions to create, copy or removed data associated with the application version. It is up to the provider to determine the appropriate action to take, if any.

In addition, the versionable application provider SSPI is also called when a non-versioned application is deleted so that the provider can perform cleanup actions.

The WebLogic Security Framework passes the versionable application provider the application identifier for the new version and the application identifier of the version used as the source of application data. When the source identifier is not supplied, the initial version of the application is being created.

Do You Need to Develop a Custom Versionable Application Provider?

The WebLogic Server out-of-the-box security providers for authorization, role mapping, and credential mapping support the application versioning SSPI. When a new version is created, all the customized roles, policies and credential maps are cloned with new resource identifiers representing the new application version. In addition, when an application version is deleted, resources associated with the deleted version are removed.

If you develop a custom security provider for authorization, role mapping, or credential mapping and need to support versioned applications, you must implement the versionable application SSPI.

How to Develop a Custom VersionableApplication Provider

If you need to support the versionable application SSPI, you can develop a custom versionable application provider by following these steps:

- Implement your custom authorization, role mapping, or credential mapping providers. All authorization, role mapping, or credential mapping providers for the security realm must support application versioning for an application to be deployed using versions.
- Create Runtime Classes Using the Appropriate SSPIs
- Generate an MBean Type Using the WebLogic MBeanMaker

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom versionable application provider by following these steps:

- Implement your custom authorization, role mapping, or credential mapping providers.
- Implement the VersionableApplication SSPI

Implement the VersionableApplication SSPI

To implement the `VersionableApplication` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following methods:

- createApplicationVersion

  ```
  void createApplicationVersion(String appIdentifier, String sourceAppIdentifier)
  ```

Marks the creation of a new application version and is called (only on the Administration Server within a WebLogic Server domain) on one server within a WebLogic Server domain at the time the version is created. The WebLogic Security Framework passes the `createApplicationVersion` method the application identifier for the new version (`appIdentifier`) and the application identifier of the version used as the source of application data (`sourceAppIdentifier`). When the source identifier is not supplied, the initial version of the application is being created.

- deleteApplication

  ```
  void deleteApplication(String appName)
  ```

  Marks the deletion of a non-versioned application and is called (only on the Administration Server within a WebLogic Server domain) at the time the application is deleted.

- deleteApplicationVersion

  ```
  void deleteApplicationVersion(String appIdentifier)
  ```

  Marks the deletion of an application version and is only called (only on the Administration Server within a WebLogic Server domain) at the time the version is deleted.

Example: Creating the Runtime Class for the Sample VersionableApplication Provider

Example 13-1 shows how the versionable application SSPI is implemented in the sample authorization provider.

Example 13-1 SimpleSampleAuthorizationProviderImpl

```
public final class SimpleSampleAuthorizationProviderImpl
  implements DeployableAuthorizationProviderV2, AccessDecision,
VersionableApplicationProvider
:
:
public void createApplicationVersion(String appId, String sourceAppId)
{
System.out.println("SimpleSampleAuthorizationProviderImpl.createApplicationVersion");
System.out.println("\tapplication identifier\t= " + appId);
System.out.println("\tsource app identifier\t= " + ((sourceAppId != null) ?
sourceAppId : "None"));
// create new policies when existing application is specified
    if (sourceAppId != null) {
      database.clonePoliciesForApplication(sourceAppId,appId);
    }

public void deleteApplicationVersion(String appId)
{
System.out.println("SimpleSampleAuthorizationProviderImpl.deleteApplicationVersion");
System.out.println("\tapplication identifier\t= " + appId);

// clear out policies for the application
database.removePoliciesForApplication(appId);
}

public void deleteApplication(String appName)
{
System.out.println("SimpleSampleAuthorizationProviderImpl.deleteApplication");
System.out.println("\tapplication name\t= " + appName);

// clear out policies for the application
database.removePoliciesForApplication(appName);
}
```

Generate an MBean Type Using the WebLogic MBeanMaker

When you generate the MBean type for your custom authorization, role mapping, and credential mapping providers, you must also implement the MBean for your versionable application provider. The ApplicationVersionerMBean is a marker interface and has no methods.

Example 13-2 shows how the SimpleSampleAuthorizer MBean Definition File (MDF) implements the ApplicationVersionerMBean MBean.

Example 13-2 Implementing the ApplicationVersionerMBean

```
<MBeanType
 Name          = "SimpleSampleAuthorizer"
 DisplayName   = "SimpleSampleAuthorizer"
 Package       = "examples.security.providers.authorization.simple"
 Extends       = "weblogic.management.security.authorization.DeployableAuthorizer"
 Implements    = "weblogic.management.security.ApplicationVersioner"
 PersistPolicy = "OnUpdate"
>
```

Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF)

Once your have run your MDF through the WebLogic MBeanMaker to generate your intermediate files, and you have edited the MBean implementation file to supply implementations for the appropriate methods within it, you need to package the MBean files and the runtime classes for the custom authorization, role mapping, or credential mapping provider, including the versionable application provider, into an MBean JAR File (MJF).

These steps are described in Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF).

Configure the Custom Versionable Application Provider

Configuring a custom versionable application provider means that you are adding the custom versionable application provider to your security realm, where it can be accessed by applications requiring application version services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers.

The steps for configuring a custom versionable application provider are described in Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

14

CertPath Providers

This chapter describes the background information you need to understand before adding certificate lookup and validation capability to your custom security providers, and provides step-by-step instructions for adding certificate lookup and validation capability to a custom security provider.
The WebLogic Security service provides a framework that finds and validates X509 certificate chains for inbound 2-way SSL, outbound SSL, application code, and WebLogic Web services. The Certificate Lookup and Validation (CLV) framework is a new security plug-in framework that finds and validates certificate chains. The framework extends and completes the JDK CertPath functionality, and allows you to create a custom CertPath provider.

This chapter includes the following sections:

Certificate Lookup and Validation Concepts

A CertPath is a JDK class that stores a certificate chain in memory. The term CertPath is also used to refer to the JDK architecture and framework that is used to locate and validate certificate chains.

There are two distinct types of providers, CertPath Validators and CertPath Builders:

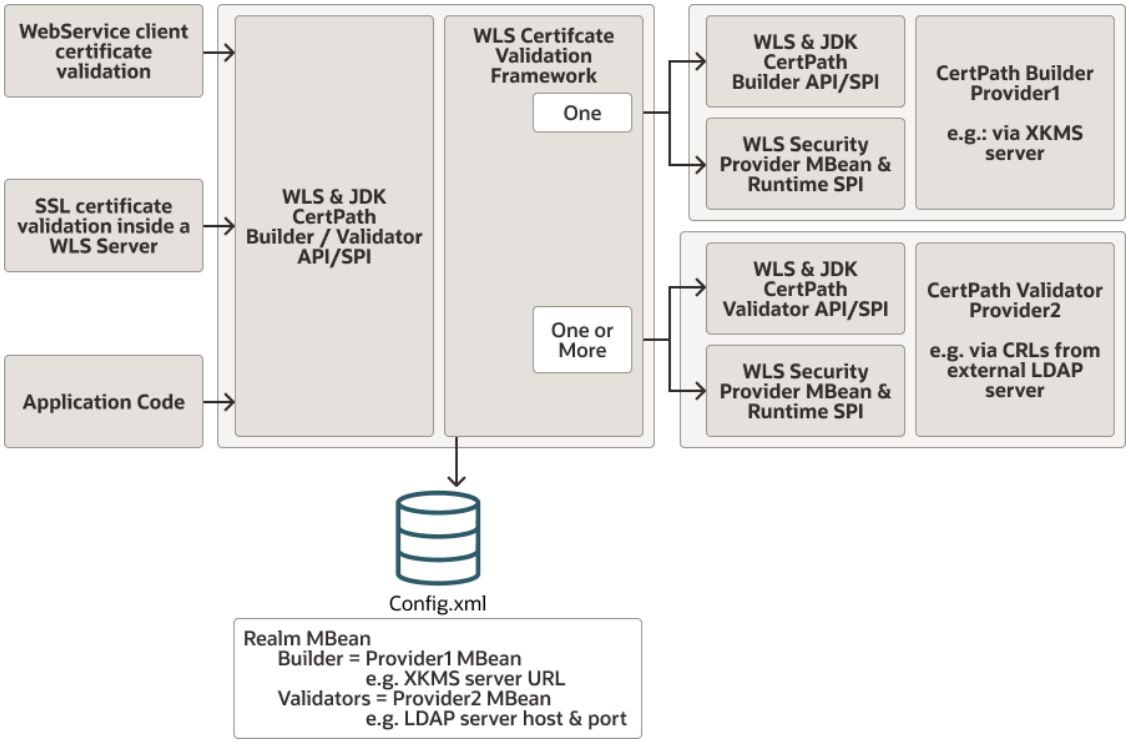
- The purpose of a certificate validator is to determine if the presented certificate chain is valid and trusted. As the CertPath Validator provider writer, you decide how to validate the certificate chain and determine whether you need to use the trusted CA's.
- The purpose of a certificate builder is to use a selector (which holds the selection criteria for finding the CertPath) to find a certificate chain. Certificate builders often to validate the certificate chain as well. As the CertPath Builder provider writer, you decide which of the four selector types you support and whether you also validate the certificate chain. You also decide how much of the certificate chain you fill in and whether you need to use the trusted CA's.

The WebLogic CertPath providers are built using both the JDK and WebLogic CertPath SPI's.

The Certificate Lookup and Validation Process

The certificate lookup and validation process is shown in Figure 14-1.

Figure 14-1 Certificate Lookup and Validation Process



Do You Need to Implement Separate CertPath Validators and Builders?

You can implement the CertPath provider in several ways:

- You can implement a CertPath Builder that performs both building and validation. In this case, you are responsible for:
  1. Implementing the Validator SPI.
  2. Implementing the Builder SPI.
  3. You must validate the certificate chain you build as part of the Builder SPI. Your provider will be called only once; you will not be called a second time specifically for validation.
  4. You decide the validation algorithm, which selectors to support, and whether to use trusted CA's.
- You can implement a CertPath Validator that performs only validation. In this case, you are responsible for:
  1. Implementing the Validator SPI.
  2. You decide the validation algorithm and whether to use trusted CA's.
- You can implement a CertPath Builder that performs only building. In this case, you are responsible for:
  1. Implementing the Builder SPI.
  2. You decide whether to validate the chain you build.
  3. You decide which selectors to support and whether to use trusted CA's.

CertPath Provider SPI MBeans

WebLogic Server includes two CertPath provider SPI MBeans, both of which extend CertPathProviderMBean:

- CertPathBuilderMBean indicates that the provider can look up certificate chains. It adds no attributes or methods. CertPathBuilder providers must implement a custom MBean that extends this MBean.
- CertPathValidatorMBean indicates that the provider can validate a certificate chain. It adds no attributes or methods. CertPathValidator providers must implement a custom MBean that extends this MBean.

Your CertPath provider, depending on its type, must extend one or both of the MBeans. A security provider that supports both building and validating should write an MBean that extends both of these MBeans, as shown in Example 14-1.

Example 14-1 Sample CertPath MBean MDF

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">

<MBeanType
Name          = "MyCertPathProvider"
DisplayName   = "MyCertPathProvider"
Package       = "com.acme"
Extends       = "weblogic.management.security.pk.CertPathBuilder"
Implements    = "weblogic.management.security.pk.CertPathValidator"
PersistPolicy = "OnUpdate"
>
<MBeanAttribute
Name          = "ProviderClassName"
Type          = "java.lang.String"
Writeable     = "false"
Default       = "&quot;com.acme.MyCertPathProviderRuntimeImpl&quot;"
/>

<MBeanAttribute
Name          = "Description"
Type          = "java.lang.String"
Writeable     = "false"
Default       = "&quot;My CertPath Provider&quot;"
/>

<MBeanAttribute
Name          = "Version"
Type          = "java.lang.String"
Writeable     = "false"
Default       = "&quot;1.0&quot;"
/>

 <!-- add custom attributes for the configuration data needed by this provider -->
<MBeanAttribute
Name          = "CustomConfigData"
Type          = "java.lang.String"
/>
```

WebLogic CertPath Validator SSPI

The WebLogic CertPath Validator SSPI has four parts:

- An MBean SSPI, described in CertPath Provider SPI MBeans.
- The JDK `CertPathValidatorSPI` interface, as described inImplement the JDK CertPathBuilderSpi and/or CertPathValidatorSpi Interfaces .
- The WebLogic Server `CertPathProvider` SSPI interface, as described in Implement the CertPath Provider SSPI.
- The JDK security `provider` that registers your `CertPathValidatorSPI` implementation with the JDK, as described in Implement the JDK Security Provider SPI.

WebLogic CertPath Builder SSPI

The WebLogic CertPath Builder SSPI has four parts:

- An MBean SSPI, described in CertPath Provider SPI MBeans.
- The JDK `CertPathBuilder`SPI interface, as described inImplement the JDK CertPathBuilderSpi and/or CertPathValidatorSpi Interfaces .
- The WebLogic Server `CertPathProvider` SSPI interface, as described in Implement the CertPath Provider SSPI.
- The JDK security `provider` that registers your `CertPathBuilder`SPI with the JDK, as described in Implement the JDK Security Provider SPI.

Relationship Between the WebLogic Server CertPath SSPI and the JDK SPI

Unlike other WebLogic Security Framework providers, your implementation of the CertPath provider relies on a tightly-coupled integration of WebLogic and JDK interfaces. This integration might best be shown in the tasks you perform to create a CertPath provider.

If you are writing a CertPath Validator, you must perform the following tasks:

1. Create a *CertPathValidatorMBean* that extends *CertPathProviderMBean*, as described in Generate an MBean Type Using the WebLogic MBeanMaker.
2. Implement the JDK `java.security.cert.CertPathValidatorSpi`, as described in Implement the JDK CertPathBuilderSpi and/or CertPathValidatorSpi Interfaces .

   Your JDK implementation will be passed a JDK `CertPathParameters` object that you can cast to a WebLogic `CertPathValidatorParametersSpi`. You can then access its WebLogic methods to get the trusted CA's and ContextHandler. You can also use it to access your WebLogic CertPath provider object.

   Use the `CertPathValidatorParametersSpi` to provide the data you need to validate the certificate chain, such as Trusted CA's, the ContextHandler, and your CertPath provider SSPI implementation, which gives access to any custom configuration data provided by your MBean, as described in Use the CertPathValidatorParametersSpi SSPI in Your CertPathValidatorSpi Implementation .

   Your WebLogic CertPath provider is important because your `CertPathValidatorSpi` implementation has no direct way to get the custom configuration data in your MBean. Your WebLogic CertPath provider can provide a proprietary mechanism to make your custom MBean data available to your JDK implementation.
3. Implement the WebLogic CertPath provider SSPI, as described in Implement the CertPath Provider SSPI. In particular, you use the `initialize` method of the CertPath provider SSPI to hook into the MBean and make its custom configuration data available to your `CertPathValidatorSpi` implementation, as shown in Example 14-2.

4. Implement a JDK security `provider` that registers your CertPathValidatorSpi implementation, as described in Implement the JDK Security Provider SPI. This coding might not be intuitive, and is called out in Example 14-5.

If you are writing a CertPath Builder, you must perform the following tasks:

1. Create a CertPathBuilderMBean that extends CertPathProviderMBean, as described in Generate an MBean Type Using the WebLogic MBeanMaker.
2. Implement the JDK `java.security.cert.CertPathBuilderSpi`, as described in Implement the JDK CertPathBuilderSpi and/or CertPathValidatorSpi Interfaces .

   Your JDK implementation will be passed a JDK `CertPathParameters` object that you can cast to a WebLogic `CertPathBuilderParametersSpi`. You can then access its WebLogic methods to get the trusted CA's, selector, and ContextHandler. You can also use it to access your WebLogic CertPath provider object.

   Use the `CertPathBuilderParametersSpi` to provide the data you need to build the CertPath, such as Trusted CA's, ContextHandler, the CertPathSelector, and your CertPath provider SSPI implementation, which gives access to any custom configuration data provided by your MBean, as described in Use the CertPathBuilderParametersSpi SSPI in Your CertPathBuilderSpi Implementation .

   Your WebLogic CertPath provider is important because your `CertPathBuilderSpi` implementation has no direct way to get the custom configuration data in your MBean. Your WebLogic CertPath provider can provide a proprietary mechanism to make your custom MBean data available to your JDK implementation.
3. Implement a WebLogic CertPath provider SSPI, as described in Implement the CertPath Provider SSPI. In particular, you use the `initialize` method of the CertPath provider SSPI to hook into the MBean and make its custom configuration data available to your `CertPathBuilderSpi` implementation, as shown in Example 14-2.
4. Implement the JDK security `provider` that registers your CertPathBuilderSpi implementation, as described in Implement the JDK Security Provider SPI. This coding might not be intuitive, and is called out in Example 14-5.

Do You Need to Develop a Custom CertPath Provider?

WebLogic Server includes a CertPath provider and the Certificate Registry.

The WebLogic Server CertPath provider is both a CertPath Builder and a CertPath Validator. The provider completes certificate paths and validates the certificates using the trusted CA configured for a particular WebLogic Server instance. It can build only chains that are self-signed or are issued by a self-signed certificate authority, which must be listed in the server's trusted CA's. If a certificate chain cannot be completed, it is invalid. The provider uses only the `EndCertificateSelector` selector.

The WebLogic Server CertPath provider also checks the signatures in the chain, ensures that the chain has not expired, and checks that one of the certificates in the chain is issued by one of the trusted CAs configured for the server. If any of these checks fail, the chain is not valid. Finally, the provider checks each certificate's basic constraints (that is, the ability of the certificate to issue other certificates) to ensure the certificate is in the proper place in the chain.

The WebLogic Server CertPath provider can be used as a CertPath Builder and a CertPath Validator in a security realm.

The WebLogic Server Certificate Registry is an out-of-the-box CertPath provider that allows the administrator to configure a list of trusted end certificates using the WebLogic Remote Console. The Certificate Registry is a builder/validator. The selection criteria can be

`EndCertificateSelector`, `SubjectDNSelector`, `IssuerDNSerialNumberSelector`, or `SubjectKeyIdentifier`. The certificate chain that is returned has only the end certificate. When it validates a chain, it makes sure only that the end certificate is registered; no further checking is done.

You can configure both the CertPath provider and the Certificate Registry. You might do this to make sure that a certificate chain is valid only if signed by a trusted CA, and that the end certificate is in the registry.

If the supplied WebLogic Server CertPath providers do not meet your needs, you can develop a custom CertPath provider.

How to Develop a Custom CertPath Provider

If the WebLogic CertPath provider or Certificate Registry does not meet your needs, you can develop a custom CertPath provider by following these steps:

Create Runtime Classes Using the Appropriate SSPIs

Before you start creating runtime classes, you should first:

- Understand the Purpose of the Provider SSPIs
- Understand the SSPI Hierarchy and Determine Whether You Will Create One or Two Runtime Classes

When you understand this information and have made your design decisions, create the runtime classes for your custom CertPath provider by completing the steps described in the following sections:

- Generate an MBean type for your custom authentication provider by completing the steps described in Generate an MBean Type Using the WebLogic MBeanMaker.
- Implement the JDK CertPathBuilderSpi and/or CertPathValidatorSpi Interfaces
- Implement the CertPath Provider SSPI
- Implement the JDK Security Provider SPI
- Use the CertPathBuilderParametersSpi SSPI in Your CertPathBuilderSpi Implementation and/or Use the CertPathValidatorParametersSpi SSPI in Your CertPathValidatorSpi Implementation

Implement the JDK CertPathBuilderSpi and/or CertPathValidatorSpi Interfaces

The `java.security.cert.CertPathBuilderSpi` interface is the Service Provider Interface (SPI) for the `CertPathBuilder` class. All `CertPathBuilder` implementations must include a class that implements this interface (`CertPathBuilderSpi`).

The `java.security.cert.CertPathValidatorSpi` interface is the Service Provider Interface (SPI) for the `CertPathValidator` class. All `CertPathValidator` implementations must include a class that implements this interface (`CertPathValidatorSpi`).

Example 14-6 shows an example of implementing the `CertPathBuilderSpi` and `CertPathValidatorSpi` interfaces.

Implement the CertPath Provider SSPI

The `CertPathProvider` SSPI interface exposes the services provided by both the JDK `CertPathValidator` and `CertPathBuilder` SPIs and allows the provider to be manipulated (initialized, started, stopped, and so on).

In particular, you use the `initialize` method of the CertPath provider SSPI to hook into the MBean and make its custom configuration data available to your `CertPathBuilderSpi` or `CertPathValidatorSpi` implementation, as shown in Example 14-2.

A more complete example is available in Example 14-6.

Example 14-2 Code Fragment: Obtaining Custom Configuration Data From MBean

```
public class MyCertPathProviderRuntimeImpl implements CertPathProvider
{
:
:
   public void initialize(ProviderMBean mBean, SecurityServices securityServices)
   {
     MyCertPathProviderMBean myMBean = (MyCertPathProviderMBean)mBean;
     description = myMBean.getDescription();
     customConfigData = myMBean.getCustomConfigData();
:
}
:
   // make my config data available to my JDK CertPathBuilderSpi and
   // CertPathValidatorSpi impls
   private String getCustomConfigData() { return customConfigData; }
}
:
static public class MyJDKCertPathBuilder extends CertPathBuilderSpi
{
:
//get my runtime implementation instance which holds the configuration
//data needed to build and validate the cert path
MyCertPathProviderRuntimeImpl runtime =
(MyCertPathProviderRuntimeImpl)params.getCertPathProvider();
String myCustomConfigData = runtime.getCustomConfigData();
```

Example 14-5 shows how to register your JDK implementation with the JDK.

To implement the `CertPathProvider` SSPI, provide implementations for the methods described in Understand the Purpose of the Provider SSPIs *and* the following methods:

- getCertPathBuilder

  ```
  public CertPathBuilder getCertPathBuilder()
  ```

  Gets a CertPath Provider's JDK CertPathBuilder that invokes your JDK `CertPathBuilderSpi` implementation, as shown in Example 14-3. A CertPathBuilder finds, and optionally validates, a certificate chain.

Example 14-3 Code Fragment: getCertPathBuilder

```
public void initialize(ProviderMBean mBean, SecurityServices securityServices)
{
:
     // get my JDK cert path impls
     try {
```

```
      certPathBuilder = CertPathBuilder.getInstance(BUILDER_ALGORITHM);
    } catch (NoSuchAlgorithmException e) { throw new AssertionError("..."); }
```

- getCertPathValidator

  ```
  public CertPathValidator getCertPathValidator()
  ```

  Gets a CertPath Provider's JDK CertPathValidator that invokes your JDK `CertPathValidatorSpi` implementation, as shown in Example 14-4. A CertPathValidator validates a certificate chain.

Example 14-4 Code Fragment: getCertPathValidator

```
public void initialize(ProviderMBean mBean, SecurityServices securityServices)
{
:
     // get my JDK cert path impls
     try {
       certPathValidator = CertPathValidator.getInstance(VALIDATOR_ALGORITHM);
     } catch (NoSuchAlgorithmException e) { throw new AssertionError("..."); }
   }
```

Implement the JDK Security Provider SPI

Implement the JDK security `provider` SPI and use it to register your `CertPathBuilderSpi` or `CertPathValidatorSpi` implementations with the JDK. Use it to register your JDK implementation in your provider's `initialize` method.

Example 14-6 shows an example of creating the runtime class for a sample CertPath provider. Example 14-5 shows the fragment from that larger example that implements the JDK security `provider`.

Example 14-5 Implementing the JDK Security Provider

```
public class MyCertPathProviderRuntimeImpl implements CertPathProvider
{
private static final String MY_JDK_SECURITY_PROVIDER_NAME = "MyCertPathProvider";
private static final String BUILDER_ALGORITHM = MY_JDK_SECURITY_PROVIDER_NAME + "CertPathBuilder";
private static final String VALIDATOR_ALGORITHM = MY_JDK_SECURITY_PROVIDER_NAME + "CertPathValidator";
:
:
   public void initialize(ProviderMBean mBean, SecurityServices securityServices)
   {
     MyCertPathProviderMBean myMBean = (MyCertPathProviderMBean)mBean;

     description = myMBean.getDescription();

     customConfigData = myMBean.getCustomConfigData();

// register my cert path impls with the JDK
// so that the CLV framework may invoke them via
// the JDK cert path apis.
if (Security.getProvider(MY_JDK_SECURITY_PROVIDER_NAME) == null) {
   AccessController.doPrivileged(
     new PrivilegedAction() {
       public Object run() {
         Security.addProvider(new MyJDKSecurityProvider());
         return null;
       }
     }
   );
}
```

```
:
// This class implements the JDK security provider that registers
// this provider's cert path builder and cert path validator implementations
// with the JDK.
private class MyJDKSecurityProvider extends Provider
 {
   private MyJDKSecurityProvider()
    {
      super(MY_JDK_SECURITY_PROVIDER_NAME, 1.0, "MyCertPathProvider JDK CertPath provider");
      put("CertPathBuilder." + BUILDER_ALGORITHM,
 "com.acme.MyPathProviderRuntimeImpl$MyJDKCertPathBuilder");
      put("CertPathValidator." + VALIDATOR_ALGORITHM,
 "com.acme.MyCertPathProviderRuntimeImpl$MyJDKCertPathValidator");
    }
 }
}
```

Use the CertPathBuilderParametersSpi SSPI in Your CertPathBuilderSpi Implementation

Your JDK implementation will be passed a JDK `CertPathParameters` object that you can cast to a WebLogic `CertPathBuilderParametersSpi`. You can then access its WebLogic methods to get the trusted CA's, selector, and ContextHandler. You can also use it to access your WebLogic CertPath provider object. The following methods are provided:

- getCertPathProvider

  ```
  CertPathProvider getCertPathProvider()
  ```

  Gets the CertPath Provider SSPI interface that exposes the services provided by a CertPath provider to the WebLogic Security Framework. In particular, you use the `initialize` method of the CertPath provider SSPI to hook into the MBean and make its custom configuration data available to your `CertPathBuilderSpi` implementation, as shown in Example 14-2.

- getCertPathSelector

  ```
  CertPathSelector getCertPathSelector()
  ```

  Gets the `CertPathSelector` interface that holds the selection criteria for finding the CertPath.

  WebLogic Server provides a set of classes in `weblogic.security.pk` that implement the `CertPathSelector` interface, one for each supported type of certificate chain lookup. Therefore, the `getCertPathSelector` method returns one of the following derived classes:

- `EndCertificateSelector` – used to find and validate a certificate chain given its end certificate.
- `IssuerDNSerialNumberSelector` – used to find and validate a certificate chain from its end certificate's issuer DN and serial number.
- `SubjectDNSelector` – used to find and validate a certificate chain from its end certificate's subject DN.
- `SubjectKeyIdentifierSelector` – used to find and validate a certificate chain from its end certificate's subject key identifier (an optional field in X509 certificates).

  Each selector class has one or more methods to retrieve the selection data and a constructor.

Your `CertPathBuilderSpi` implementation decides which selectors it supports. The `CertPathBuilderSpi` implementation must use the `getCertPathSelector` method of the `CertPathBuilderParametersSpi` SSPI to get the `CertPathSelector` that holds the selection criteria for finding the CertPath. If your `CertPathBuilderSpi` implementation supports that type of selector, it then uses the selector to build and validate the chain. Otherwise, it must throw an InvalidAlgorithmParameterException, which is propagated back to the caller.

- getContext()

  ```
  ContextHandler getContext()
  ```

  Gets a ContextHandler that may pass in extra parameters that can be used for building and validating the CertPath.

- getTrustedCAs()

  ```
  X509Certificate[] getTrustedCAs()
  ```

  Gets a list of trusted certificate authorities that may be used for building the certificate chain. If your `CertPathBuilderSpi` implementation needs Trusted CA's to build the chain, it should use these Trusted CA's.

- clone

  ```
  Object clone()
  ```

  This interface is not cloneable.

Use the CertPathValidatorParametersSpi SSPI in Your CertPathValidatorSpi Implementation

Your JDK implementation will be passed a JDK `CertPathParameters` object that you can cast to a WebLogic `CertPathValidatorParametersSpi`. You can then access its WebLogic methods to get the trusted CA's and ContextHandler. You can also use it to access your WebLogic CertPath provider object. The CLV framework ensures that the certificate chain passed to the validator SPI is in order (starting at the end certificate), and that each cert has signed the next. The following methods are provided:

- getCertPathProvider

  ```
  CertPathProvider getCertPathProvider()
  ```

  Gets the CertPath Provider SSPI interface that exposes the services provided by a CertPath provider to the WebLogic Security Framework. In particular, you use the `initialize` method of the CertPath provider SSPI to hook into the MBean and make its custom configuration data available to your `CertPathValidatorSpi` implementation, as shown in Example 14-2.

- getContext()

  ```
  ContextHandler getContext()
  ```

  Gets a ContextHandler that may pass in extra parameters that can be used for building and validating the CertPath.

  SSL performs some built-in validation before it calls one or more CertPathValidator objects to perform additional validation. A validator can reduce the amount of validation it must do by discovering what validation has already been done.

For example, the WebLogic CertPath Provider performs the same validation that SSL does, and there is no need to duplicate that validation when invoked by SSL. Therefore, SSL puts some information into the context it hands to the validators to indicate what validation has already occurred. The `weblogic.security.SSL.SSLValidationConstants CHAIN_PREVALIDATED_BY_SSL` field is a Boolean that indicates whether SSL has pre-validated the certificate chain. Your application code can test this field, which is set to true if SSL has pre-validated the certificate chain, and is false otherwise.

- getTrustedCAs()

  ```
  X509Certificate[] getTrustedCAs()
  ```

  Gets a list of trusted certificate authorities that may be used for validating the certificate chain. If your CertPathBuilderSpi implementation needs Trusted CA's to validate the chain, it should use these Trusted CA's.

- clone

  ```
  Object clone()
  ```

  This interface is not cloneable.

Returning the Builder or Validator Results

Your JDK CertPathBuilder or CertPathValidator implementation must return an object that implements the `java.security.cert.CertPathValidatorResult` or `java.security.cert.CertPathValidatorResult` interface.

You can write your own results implementation or you can use the WebLogic Server convenience routines.

WebLogic Server provides two convenience results-implementation classes, `WLSCertPathBuilderResult` and `WLSCertPathValidatorResult`, both of which are located in `weblogic.security.pk`, that you can use to return instances of `java.security.cert.CertPathValidatorResult` or `java.security.cert.CertPathValidatorResult`.

> **Note**
>
> The results you return are not passed through the WebLogic Security framework.

Example: Creating the Sample Cert Path Provider

Example 14-6 shows an example CertPath builder/validator provider. The example includes extensive comments that explain the code flow.

Example 14-1 shows the CertPath MBean that Example 14-6 uses.

Example 14-6 Creating the Sample Cert Path Provider

```
package com.acme;

import weblogic.management.security.ProviderMBean;
import weblogic.security.pk.CertPathSelector;
import weblogic.security.pk.SubjectDNSelector;
import weblogic.security.pk.WLSCertPathBuilderResult;
import weblogic.security.pk.WLSCertPathValidatorResult;
import weblogic.security.service.ContextHandler;
```

```
import weblogic.security.spi.CertPathBuilderParametersSpi;
import weblogic.security.spi.CertPathProvider;
import weblogic.security.spi.CertPathValidatorParametersSpi;
import weblogic.security.spi.SecurityServices;
import weblogic.security.SSL.SSLValidationConstants;

import java.security.InvalidAlgorithmParameterException;
import java.security.NoSuchAlgorithmException;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.security.Provider;
import java.security.Security;
import java.security.cert.CertPath;
import java.security.cert.CertPathBuilder;
import java.security.cert.CertPathBuilderResult;
import java.security.cert.CertPathBuilderSpi;
import java.security.cert.CertPathBuilderException;
import java.security.cert.CertPathParameters;
import java.security.cert.CertPathValidator;
import java.security.cert.CertPathValidatorResult;
import java.security.cert.CertPathValidatorSpi;
import java.security.cert.CertPathValidatorException;
import java.security.cert.X509Certificate;

public class MyCertPathProviderRuntimeImpl implements CertPathProvider
{
   private static final String MY_JDK_SECURITY_PROVIDER_NAME = "MyCertPathProvider";
   private static final String BUILDER_ALGORITHM = MY_JDK_SECURITY_PROVIDER_NAME + "CertPathBuilder";
   private static final String VALIDATOR_ALGORITHM = MY_JDK_SECURITY_PROVIDER_NAME + "CertPathValidator";

   // Used to invoke my JDK cert path builder / validator implementations
   private CertPathBuilder   certPathBuilder;
   private CertPathValidator certPathValidator;

   // remember my custom configuration data from my mbean
   private String customConfigData;

   private String description;

   public void initialize(ProviderMBean mBean, SecurityServices securityServices)
   {
     MyCertPathProviderMBean myMBean = (MyCertPathProviderMBean)mBean;

     description = myMBean.getDescription();

     customConfigData = myMBean.getCustomConfigData();

     // register my cert path impls with the JDK
     // so that the CLV framework may invoke them via
     // the JDK cert path apis.
     if (Security.getProvider(MY_JDK_SECURITY_PROVIDER_NAME) == null) {
       AccessController.doPrivileged(
         new PrivilegedAction() {
           public Object run() {
            Security.addProvider(new MyJDKSecurityProvider());
             return null;
           }
         }
       );
     }

     // get my JDK cert path impls
```

```
    try {
      certPathBuilder = CertPathBuilder.getInstance(BUILDER_ALGORITHM);
    } catch (NoSuchAlgorithmException e) { throw new AssertionError("..."); }

    try {
      certPathValidator = CertPathValidator.getInstance(VALIDATOR_ALGORITHM);
    } catch (NoSuchAlgorithmException e) { throw new AssertionError("..."); }
  }

  public void             shutdown          () {                     }
  public String           getDescription       () { return description;    }
  public CertPathBuilder  getCertPathBuilder   () { return certPathBuilder;}
  public CertPathValidator getCertPathValidator () { return certPathValidator;}

  // make my config data available to my JDK CertPathBuilderSpi and
  // CertPathValidatorSpi impls
  private String getCustomConfigData() { return customConfigData; }

  /**
  * This class contains JDK cert path builder implementation for this provider.
  */

  static public class MyJDKCertPathBuilder extends CertPathBuilderSpi
   {
    public CertPathBuilderResult
     engineBuild(CertPathParameters genericParams)
      throws CertPathBuilderException, InvalidAlgorithmParameterException
    {

  // narrow the CertPathParameters to the WLS ones so we can get the
  // data needed to build and validate the cert path
  if (!(genericParams instanceof CertPathBuilderParametersSpi)) {
    throw new InvalidAlgorithmParameterException("The CertPathParameters must be a
weblogic.security.pk.CertPathBuilderParametersSpi instance.");
  }

  CertPathBuilderParametersSpi params = (CertPathBuilderParametersSpi)genericParams;

  // get my runtime implementation instance which holds the configuration
  // data needed to build and validate the cert path
  MyCertPathProviderRuntimeImpl runtime = (MyCertPathProviderRuntimeImpl)params.getCertPathProvider();
  String myCustomConfigData = runtime.getCustomConfigData();

  // get the selector which indicates which cert path the caller wants built.
  // it can be an EndCertificateSelector, SubjectDNSelector,
  // IssuerDNSerialNumberSelector
  // or a SubjectKeyIdentifier.
  CertPathSelector genericSelector = params.getCertPathSelector();

  // decide which kinds of selectors this builder wants to support.

  if (genericSelector instanceof SubjectDNSelector) {

   // get the subject dn of the end certificate of the cert path the caller
  // wants built
   SubjectDNSelector selector = (SubjectDNSelector)genericSelector;
   String subjectDN = selector.getSubjectDN();

  // if your implementation requires trusted CAs, get them.
  // otherwise, ignore them.  that is, it's a quality of service
  // issue whether or not you require trusted CAs.
  X509Certificate[] trustedCAs = params.getTrustedCAs();
```

```
   // if your implementation requires looks for extra data in
   // the context handler, get it.  otherwise ignore it.
   ContextHandler context = params.getContext();
   if (context != null) {
   // ...
   }

   // use my custom configuration data (ie. myCustomConfigData),
   // the trusted CAs (if applicable to my implementation),
   // the context (if applicable to my implementation),
   // and the subject DN to build and validate the cert path
   CertPath certpath = ...
   // or X509Certificate[] chain = ...

   // if not found, throw an exception:
   if (...) {
    throw new CertPathBuilderException("Could not build a cert path for " + subjectDN);
   }

   // if not valid, throw an exception:
   if (...) {
    throw new CertPathBuilderException("Could not validate the cert path for " + subjectDN);
   }

   // if found and valid, return the cert path.
   // for convenience, use the WLSCertPathBuilderResult class
   return new WLSCertPathBuilderResult(certpath);
   // or return new WLSCertPathBuilderResult(chain);

   } else {

   // the caller passed in a selector that my implementation does not support
   throw new InvalidAlgorithmParameterException("MyCertPathProvider only
   supports weblogoic.security.pk.SubjectDNSelector");
   }
  }
}

   /**
   * This class contains JDK cert path validator implementation for this provider.
   */

   static public class MyJDKCertPathValidator extends CertPathValidatorSpi
   {
     public CertPathValidatorResult
      engineValidate(CertPath certPath, CertPathParameters genericParams)
       throws CertPathValidatorException, InvalidAlgorithmParameterException
     {

   // narrow the CertPathParameters to the WLS ones so we can get the
   // data needed to build and validate the cert path
   if (!(genericParams instanceof CertPathValidatorParametersSpi)) {
     throw new InvalidAlgorithmParameterException("The CertPathParameters must be a
 weblogic.security.pk.CertPathValidatorParametersSpi instance.");
   }

     CertPathValidatorParametersSpi params = (CertPathValidatorParametersSpi)genericParams;


     // get my runtime implementation instance which holds the configuration
     // data needed to build and validate the cert path
```

```
     MyCertPathProviderRuntimeImpl runtime = (MyCertPathProviderRuntimeImpl)params.getCertPathProvider();
     String myCustomConfigData = runtime.getCustomConfigData();

     // if your implementation requires trusted CAs, get them.
     // otherwise, ignore them.  that is, it's a quality of service
     // issue whether or not you require trusted CAs.
     X509Certificate[] trustedCAs = params.getTrustedCAs();

     // if your implementation requires looks for extra data in
     // the context handler, get it.  otherwise ignore it.
     ContextHandler context = params.getContext();
     if (context != null) {
       // ...
     }

     // The CLV framework has already done some minimal validation
     // on the cert path before sending it to your provider:
     //   1) the cert path is not empty
     //   2) the cert path starts with the end cert
     //   3) each certificate in the cert path was issued and
     //      signed by the next certificate in the chain
     //      So, your validator can rely on these checks having
     //      already been performed (vs your validator needing to
     //      do these checks too).

   // Use my custom configuration data (ie. myCustomConfigData),
   // the trusted CAs (if applicable to my implementation),
   // and the context (if applicable to my implementation)
   // to validate the cert path

   // if not valid, throw an exception:
   if (...) {
     throw new CertPathValidatorException("Could not validate the cerpath " + certPath);
   }
   // if valid, return success

   // For convenience, use the WLSCertPathValidatorResult class

   return new WLSCertPathValidatorResult();
  }
}

   // This class implements the JDK security provider that registers this    // provider's
   // cert path builder and cert path validator implementations with the JDK.
   private class MyJDKSecurityProvider extends Provider
   {
   private MyJDKSecurityProvider()
   {
     super(MY_JDK_SECURITY_PROVIDER_NAME, 1.0, "MyCertPathProvider JDK CertPath provider");
     put("CertPathBuilder." + BUILDER_ALGORITHM,
"com.acme.MyPathProviderRuntimeImpl$MyJDKCertPathBuilder");
     put("CertPathValidator." + VALIDATOR_ALGORITHM,
"com.acme.MyCertPathProviderRuntimeImpl$MyJDKCertPathValidator");
   }
  }
}
```

Configure the Custom CertPath Provider

Configuring a custom CertPath provider means that you are adding the custom CertPath provider to your security realm, where it can be accessed by applications requiring CertPath services.

Configuring custom security providers is an administrative task, but it is a task that may also be performed by developers of custom security providers.

> **Note**
>
> The steps for configuring a custom CertPath provider are described under Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.

A

MBean Definition File (MDF) Element Syntax

This appendix describes the elements and attributes that are available for use in a valid MBean Definition File (MDF). An **MBean Definition File (MDF)** is an input file to the WebLogic MBeanMaker utility, which uses the file to create an MBean type for managing a custom security provider. An MDF must be formatted as a well-formed and valid XML file that describes a single MBean type.
This appendix includes the following sections:

The MBeanType (Root) Element

All MDFs must contain exactly one root element called `MBeanType`, which has the following syntax:

```
<MBeanType Name= string   optional_attributes>
     subelements
</MBeanType>
```

The `MBeanType` element must include a Name attribute, which specifies the internal, programmatic name of the MBean type. (To specify a name that is visible in a user interface, use the DisplayName attribute.) Other attributes are optional.

The following is a simplified example of an `MBeanType` (root) element:

```
<MBeanType Name="MyMBean" Package="com.mycompany">
   <MBeanAttribute Name="MyAttr" Type="java.lang.String" Default="Hello World"/>
</MBeanType>
```

Attributes specified in the `MBeanType` (root) element apply to the entire set of MBeans instantiated from that MBean type. To override attributes for specific MBean instances, you need to specify attributes in the `MBeanAttribute` subelement. See The MBeanAttribute Subelement.

Table A-1 describes the attributes available to the `MBeanType` (root) element. The JMX Specification/Oracle Extension column indicates whether the attribute is an Oracle extension to the JMX specification or a standard JMX attribute. Note that Oracle extensions might not function on other Java EE Web servers.

Table A-1 MBeanType (Root) Element Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
`Abstract`	Oracle Extension	`true`/`false`	A `true` value specifies that the MBean type cannot be instantiated (like any abstract Java class), though other MBean types can inherit its attributes and operations. If you specify `true`, you must create other non-abstract MBean types for carrying out management tasks. If you do not specify a value for this attribute, the assumed value is `false`.
`Deprecated`	Oracle Extension	`true`/`false`	Indicates that the MBean type is deprecated. This information appears in the generated Java source, and is also placed in the `ModelMBeanInfo` object for possible use by a management application. If you do not specify this attribute, the assumed value is `false`.
`Description`	JMX Specification	*String*	An arbitrary string associated with the MBean type that appears in various locations, such as the Javadoc for generated classes. There is no default or assumed value. **Note:** To specify a description that is visible in a user interface, use the `DisplayName` attribute.
`DisplayName`	JMX Specification	*String*	The name that a user interface displays to identify instances of MBean types. For an instance of type X, the default `DisplayName` is `instance of type X`. This value is typically overridden when instances are created.
`Extends`	Oracle Extension	*Pathname*	A fully qualified MBean type name that this MBean type extends.
`Implements`	Oracle Extension	*Comma-separated list*	A comma-separated list of fully qualified MBean type names that this MBean type implements. See also Extends.
`Name`	JMX Specification	*String*	Mandatory attribute that specifies the internal, programmatic name of the MBean type.
`Package`	Oracle Extension	*String*	Specifies the package name of the MBean type and determines the location of the class files that the WebLogic MBeanMaker creates. If you do not specify this attribute, the MBean type is placed in the Java default package. **Note:** MBean type names can be the same as long as the package name varies.

Table A-1 (Cont.) MBeanType (Root) Element Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
`PersistPolicy`	JMX Specification	`/OnUpdate`	Specifies how persistence will occur: `OnUpdate`. The attribute is stored every time the attribute is updated. Note: When specified in the MBeanType element, this value overrides any setting within an individual MBeanAttribute subelement.

The MBeanAttribute Subelement

You must supply one instance of an `MBeanAttribute` subelement for each attribute in your MBean type. The `MBeanAttribute` subelement must be formatted as follows:

```
<MBeanAttribute Name=string   optional_attributes />
```

The `MBeanAttribute` subelement must include a Name attribute, which specifies the internal, programmatic name of the Java attribute in the MBean type. (To specify a name that is visible in a user interface, use the DisplayName attribute.) Other attributes are optional.

The following is a simplified example of an `MBeanAttribute` subelement within an `MBeanType` element:

```
<MBeanType Name="MyMBean" Package="com.mycompany">
    <MBeanAttribute Name= "WhenToCache"
     Type="java.lang.String"
     LegalValues="'cache-on-reference','cache-at-initialization','cache-never'"
     Default= "cache-on-reference"
    />
</MBeanType>
```

Attributes specified in an `MBeanAttribute` subelement apply to a specific MBean instance. To set attributes for the entire set of MBeans instantiated from an MBean type, you need to specify attributes in the `MBeanType` (root) element. See The MBeanType (Root) Element.

Table A-2 describes the attributes available to the `MBeanAttribute` subelement. The JMX Specification/Oracle Extension column indicates whether the attribute is an Oracle extension to the JMX specification. Note that Oracle extensions might not function on other Java EE Web servers.

Table A-2 MBeanAttribute Subelement Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
`Default`	JMX Specification	*String*	The value to be returned if the `MBeanAttribute` subelement does not provide a getter method or a cached value. The string represents a Java expression that must evaluate to an object of a type that is compatible with the provided data type for this attribute. If you do not specify this attribute, the assumed value is `null`. If you use this assumed value, and if you set the LegalNull attribute to `false`, then an exception is thrown by WebLogic MBeanMaker and WebLogic Server.
`Deprecated`	Oracle Extension	`true`/`false`	Indicates that the MBean attribute is deprecated. This information appears in the generated Java source, and is also placed in the `ModelMBeanInfo` object for possible use by a management application. If you do not specify this attribute, the assumed value is `false`.
`Description`	JMX Specification	*String*	An arbitrary string associated with the MBean attribute that appears in various locations, such as the Javadoc for generated classes. There is no default or assumed value. **Note:** To specify a description that is visible in a user interface, use the DisplayName attribute.
`Dynamic`	Oracle Extension	true/false	Changes made to dynamic MBeans take effect without rebooting the server. By default, all custom security provider MBean attributes are non-dynamic. Note that in 8.1 and 7.0, all custom security provider MBean attributes were dynamic.
`Encrypted`	Oracle Extension	`true`/`false`	A `true` value indicates that this MBean attribute will be encrypted when it is set. If you do not specify this attribute, the assumed value is `false`.

Table A-2 (Cont.) MBeanAttribute Subelement Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
`InterfaceType`	Oracle Extension	`String`	Classname of an interface to be used instead of the MBean interface generated by the WebLogic MBeanMaker. `InterfaceType` can be • int • long • float • double • char • byte Do not specify if Type is `java.lang.String`, `java.lang.String[]`, or `java.lang.Properties`.
`IsIs`	JMX Specification	`true/false`	Specifies whether a generated Java interface uses the JMX `is<AttributeName>` method to access the boolean value of the MBean attribute (as opposed to the `get<AttributeName>` method). If you do not specify this attribute, the assumed value is `false`.
`LegalNull`	Oracle Extension	`true/false`	Specifies whether `null` is an allowable value for the current `MBeanAttribute` subelement. If you do not specify this attribute, the assumed value is `true`.
`LegalValues`	Oracle Extension	*Comma-separated list*	Specifies a fixed set of allowable values for the current `MBeanAttribute` subelement. If you do not specify this attribute, the MBean attribute allows any value of the type that is specified by the Type attribute. **Note:** The items in the list must be convertible to the data type that is specified by the subelement's `Type` attribute.
`Max`	Oracle Extension	*Integer*	For numeric MBean attribute types only, provides a numeric value that represents the inclusive maximum value for the attribute. If you do not specify this attribute, the value can be as large as the data type allows.
`Min`	Oracle Extension	*Integer*	For numeric MBean attribute types only, provides a numeric value which represents the inclusive minimum value for the attribute. If you do not specify this attribute, the value can be as small as the data type allows.

Table A-2 (Cont.) MBeanAttribute Subelement Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
`Name`	JMX Specification	*String*	Mandatory attribute that specifies the internal, programmatic name of the MBean attribute.
`Type`	JMX Specification	`Java class name`	The fully qualified classname of the data type of this attribute. This corresponding class must be available on the classpath. If you do not specify this attribute, the assumed value is `java.lang.String`. Type can be • `java.lang.Integer` • `java.lang.Integer[]` • `java.lang.Long` • `java.lang.Long[]` • `java.lang.Float` • `java.lang.Float[]` • `java.lang.Double` • `java.lang.Double[]` • `java.lang.Char` • `java.lang.Char[]` • `java.lang.Byte` • `java.lang.Byte[]` • `java.lang.String` • `java.lang.String[]` • `java.util.Properties`
`Writeable`	JMX Specification	`true/false`	A `true` value allows the MBean API to set an `MBeanAttribute`'s value. If you do not specify this attribute in `MBeanType` or `MBeanAttribute`, the assumed value is `true`. When specified in the `MBeanType` element, this value is considered the default for individual `MBeanAttribute` subelements.

The MBeanConstructor Subelement

`MBeanConstructor` subelements are not currently used by the WebLogic MBeanMaker, but are supported for compliance with the Java Management eXtensions specification and upward compatibility. Therefore, attribute details for the `MBeanConstructor` subelement (and its associated `MBeanConstructorArg` subelement) are omitted from this documentation.

The MBeanOperation Subelement

You must supply one instance of an `MBeanOperation` subelement for each operation (method) that your MBean type supports. The `MBeanOperation` must be formatted as follows:

```
<MBeanOperation Name=string    optional_attributes >
     <MBeanOperationArg Name=string   optional_attributes />
</MBeanOperation>
```

The `MBeanOperation` subelement must include a Name attribute, which specifies the internal, programmatic name of the operation. (To specify a name that is visible in a user interface, use the DisplayName attribute.) Other attributes are optional.

Within the `MBeanOperation` element, you must supply one instance of an `MBeanOperationArg` subelement for each argument that your operation (method) uses. The `MBeanOperationArg` must be formatted as follows:

`<MBeanOperationArg Name=`*`string optional_attributes`*` />`

The `Name` attribute must specify the name of the operation. The only optional attribute for `MBeanOperationArg` is `Type`, which provides the Java class name that specifies behavior for a specific type of Java attribute. If you do not specify this attribute, the assumed value is `java.lang.String`.

The following is a simplified example of an `MBeanOperation` and `MBeanOperationArg` subelement within an `MBeanType` element:

```
<MBeanType Name="MyMBean" Package="com.mycompany">
   <MBeanOperation
    Name= "findParserSelectMBeanByKey"
    ReturnType="XMLParserSelectRegistryEntryMBean"
    Description="Given a public ID, system ID, or root element tag, returns the
 object name of the corresponding XMLParserSelectRegistryEntryMBean."
   >
      <MBeanOperationArg Name="publicID" Type="java.lang.String"/>
      <MBeanOperationArg Name="systemID" Type="java.lang.String"/>
      <MBeanOperationArg Name="rootTag" Type="java.lang.String"/>
   </MBeanOperation>
</MBeanType>
```

Table A-3 describes the attributes available to the `MBeanOperation` subelement. The JMX Specification/Oracle Extension column indicates whether the attribute is an Oracle extension to the JMX specification. Note that Oracle extensions might not function on other Java EE Web servers.

Table A-3 MBeanOperation Subelement Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
`Deprecated`	Oracle Extension	`true`/`false`	Indicates that the MBean operation is deprecated. This information appears in the generated Java source, and is also placed in the `ModelMBeanInfo` object for possible use by a management application. If you do not specify this attribute, the assumed value is `false`.

Table A-3 (Cont.) MBeanOperation Subelement Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
`Description`	JMX Specification	*String*	An arbitrary string associated with the MBean operation that appears in various locations, such as the Javadoc for generated classes. There is no default or assumed value. **Note:** To specify a description that is visible in a user interface, use the DisplayName attribute.
`Name`	JMX Specification	*String*	Mandatory attribute that specifies the internal, programmatic name of the MBean operation.
`ReturnType`	JMX Specification	*String*	A string containing the fully qualified classname of the Java object returned by the operation being described. `ReturnType` can be void or the following: • `int` • `int[]` • `long` • `long[]` • `float` • `float[]` • `double` • `double[]` • `char` • `char[]` • `byte` • `byte[]` • `java.lang.String` • `java.lang.String[]` • `java.util.Properties`

Table A-4 describes the attributes available to the `MBeanOperationArg` subelement. The JMX Specification/Oracle Extension column indicates whether the attribute is an Oracle extension to the JMX specification. Note that Oracle extensions might not function on other Java EE Web servers.

Table A-4 MBeanOperationArg Subelement Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
`Description`	JMX Specification	*String*	An arbitrary string associated with the MBean operation argument that appears in various locations, such as the Javadoc for generated classes. There is no default or assumed value.
`Name`	JMX Specification	*String*	Mandatory attribute that specifies the name of the argument.

Table A-4 (Cont.) MBeanOperationArg Subelement Attributes

Attribute	JMX Specification /Oracle Extension	Allowed Values	Description
Type	JMX Specification	*String*	The type of the MBean operation argument. If you do not specify this attribute, the assumed value is `java.lang.String`. Type can be • `int` • `int[]` • `long` • `long[]` • `float` • `float[]` • `double` • `double[]` • `char` • `char[]` • `byte` • `byte[]` • `java.lang.String` • `java.lang.String[]` • `java.util.Properties`

MBean Operation Exceptions

Your MBean Definition Files (MDFs) must use only JDK exception types or `weblogic.management.utils` exception types. The following is a code fragment from Example A-1 that shows the use of an `MBeanException` within an `MBeanOperation` subelement:

```
<MBeanOperation
Name = "registerPredicate"
ReturnType = "void"
Description = "Registers a new predicate with the specified class name."
>
<MBeanOperationArg
Name = "predicateClassName"
Type = "java.lang.String"
Description = "The name of the Java class that implements the predicate."
/>
<MBeanException>weblogic.management.utils.InvalidPredicateException</MBeanException>
<MBeanException>weblogic.management.utils.AlreadyExistsException</MBeanException>
</MBeanOperation>
```

Examples: Well-Formed and Valid MBean Definition Files (MDFs)

Example A-1 and Example A-2 provide examples of MBean Definition Files (MDFs) that use many of the attributes described in this Appendix. Example A-1 shows the MDF used to generate an MBean type that manages predicates and reads data about predicates and their arguments.Example A-2 shows the MDF used to generate the MBean type for the WebLogic (default) Authorization provider.

Example A-1 PredicateEditor.xml

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">
<MBeanType
Name = "PredicateEditor"
Package = "weblogic.security.providers.authorization"
Implements = "weblogic.security.providers.authorization.PredicateReader"
PersistPolicy = "OnUpdate"
Abstract = "false"
Description = "This MBean manages predicates and reads data about predicates and their
arguments.&lt;p&gt;"
>
<MBeanOperation
Name = "registerPredicate"
ReturnType = "void"
Description = "Registers a new predicate with the specified class name."
>
<MBeanOperationArg
Name = "predicateClassName"
Type = "java.lang.String"
Description = "The name of the Java class that implements the predicate."
/>
     <MBeanException>weblogic.management.utils.InvalidPredicateException</MBeanException>
<MBeanException>weblogic.management.utils.AlreadyExistsException</MBeanException>
</MBeanOperation>

<MBeanOperation
Name = "unregisterPredicate"
ReturnType = "void"
Description = "Unregisters the currently registered predicate."   >

<MBeanOperationArg
Name = "predicateClassName"
Type = "java.lang.String"
Description = "The name of the Java class that implements predicate to be unregistered."
/>
<MBeanException>weblogic.management.utils.NotFoundException</MBeanException>
</MBeanOperation>
</MBeanType>
```

Example A-2 DefaultAuthorizer.xml

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">
<MBeanType
Name = "DefaultAuthorizer"
DisplayName = "DefaultAuthorizer"
Package = "weblogic.security.providers.authorization"
Extends ="weblogic.management.security.authorization.DeployableAuthorizer"
Implements = "weblogic.management.security.authorization.PolicyEditor,
weblogic.security.providers.authorization.PredicateEditor"
PersistPolicy = "OnUpdate"
Description = "This MBean represents configuration attributes
for the WebLogic Authorization provider. &lt;p&gt;"
>
<MBeanAttribute
Name = "ProviderClassName"
Type = "java.lang.String"
Writeable = "false"
Default"&quot;weblogic.security.providers.authorization.DefaultAuthorizationProviderImpl&
quot;"
Description = "The name of the Java class used to load the WebLogic
```

```
Authorization provider."
/>
<MBeanAttribute
Name = "Description"
Type = "java.lang.String"
Writeable = "false"
Default = "&quot;Weblogic Default Authorization Provider&quot;"    Description =
"A short description of the WebLogic Authorization provider."  />
<MBeanAttribute
Name = "Version"
Type = "java.lang.String"
Writeable = "false"
Default = "&quot;1.0&quot;"
Description = "The version of the WebLogic Authorization provider."
/>
</MBeanType>
```

B

Generate an MBean Type Using the WebLogic MBeanMaker

This appendix explains how to create the MBean type for your custom security provider. This appendix includes the following sections:

Overview of Steps

Before you start generating an MBean type for your custom security provider, you should first:

- General Architecture of a Security Provider
- Security Services Provider Interfaces (SSPIs)
- Security Service Provider Interface (SSPI) MBeans
- Security Data Migration
- Management Utilities Available to Developers of Security Providers

When you understand this information and have made your design decisions, create the MBean type for your custom security provider by completing the following steps:

1. Create an MBean Definition File (MDF)
2. Use the WebLogic MBeanMaker to Generate the MBean Type
3. Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF)
4. Install the MBean Type Into the WebLogic Server Environment

> **Note**
>
> Several sample security providers are available to illustrate how to perform these steps.
>
> All instructions provided in this section assume that you are working in a Windows environment.

Create an MBean Definition File (MDF)

To create an MBean Definition File (MDF), follow these steps:

1. Copy the MDF for the sample security provider to a text file.

   For each of the sample security providers, note the following MDF file names:

Table B-1 MDF File Name

Sample Security Provider Type	MDF File Name
Authentication provider	`SimpleSampleAuthenticator.xml`
Identity Assertion provider	`SampleIdentityAsserter.xml`
Authorization provider	`SimpleSampleAuthorizer.xml`
Adjudication provider	There is currently no sample adjudication provider, but you can use the MDF file for the sample authentication provider, `SimpleSampleAuthenticator.xml`.
Role Mapping provider	`SimpleSampleRoleMapper.xml`
Auditing provider	`SampleAuditor.xml`
Credential Mapping provider	There is currently no sample credential mapping provider, but you can use the MDF file for the sample authentication provider, `SimpleSampleAuthenticator.xml`.
CertPath provider	There is currently no sample CertPath provider, but you can use the MDF file for the sample authentication provider, `SimpleSampleAuthenticator.xml`.

2. Modify the content of the `<MBeanType>` and `<MBeanAttribute>` elements in your MDF so that they are appropriate for your custom security provider.

   Note the following:

   - If you are creating a custom identity assertion provider, consider the following fragment to set the `Base64DecodingRequired` attribute to `false`:

```
<MBeanAttribute
  Name         = "Base64DecodingRequired"
  Type         = "boolean"
  Writeable    = "false"
  Default      = "false"
  Description  = "See MyIdentityAsserter-doc.xml."
/>
```

   - If you are creating a custom CertPath provider, you need to extend or implement CertPathBuilderMBean or CertPathValidatorMBean.

3. Add any custom attributes and operations (that is, additional `<MBeanAttribute>` and `<MBeanOperation>` elements) to your MDF.

4. Save the file.

> **Note**
>
> A complete reference of MDF element syntax is available in MBean Definition File (MDF) Element Syntax.

Use the WebLogic MBeanMaker to Generate the MBean Type

Once you create your MDF, you are ready to run it through the WebLogic MBeanMaker. The WebLogic MBeanMaker is currently a command-line utility that takes as its input an MDF, and generates a set of intermediate Java files, including the following:

- An MBean interface
- An MBean implementation
- An associated MBean information file

Together, these intermediate files form the **MBean type** for your custom security provider.

The instructions for generating an MBean type differ based on the design of your custom security provider. Follow the instructions that are appropriate to your situation:

- No Custom Operations
- No Optional SSPI MBeans *and* No Custom Operations
- Optional SSPI MBeans or Custom Operations

No Custom Operations

This section applies to custom adjudication, role mapping, and auditing providers.

If the MDF for your custom security provider does not include any custom operations, complete the following steps:

1. Create a new DOS shell.
2. Type the following command:

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   In the preceding command:

   - The `-DMDF` flag indicates that the WebLogic MBeanMaker should translate the MDF into code.
   - *xmlFile* represents the XML MBean description file (MDF).
   - *filesdir* represents the location where the WebLogic MBeanMaker places the intermediate files for the MBean type.

   Whenever *xmlfile* is provided, a new set of output files is generated.

   Each time you use the `-DcreateStubs=true` flag, it overwrites any existing MBean implementation file.

   Whenever *`xmlfile`* is provided, a new set of output files is generated.

   Each time you use the `-DcreateStubs=true` flag, it overwrites any existing MBean implementation file.

> **Note**
>
> The WebLogic MBeanMaker processes one MDF at a time. Therefore, you may have to repeat this process if you have multiple MDFs for a given security provider type (for example, multiple adjudication providers).

3. Proceed to Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF).

No Optional SSPI MBeans *and* No Custom Operations

This section applies to the following custom security provider types:

- Authentication providers
- Identity assertion providers
- Authorization providers
- Credential mapping providers
- CertPath providers

If the MDF for your custom security provider does not implement any optional SSPI MBeans *and* does not include any custom operations, complete the following steps:

1. Create a new DOS shell.
2. Type the following command:

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
    weblogic.management.commo.WebLogicMBeanMaker
   ```

   In the preceding command:

   - The `-DMDF` flag indicates that the WebLogic MBeanMaker should translate the MDF into code.
   - *xmlFile* represents the XML MBean description file (MDF).
   - *filesdir* represents the location where the WebLogic MBeanMaker places the intermediate files for the MBean type.

   Whenever *xmlfile* is provided, a new set of output files is generated.

   Each time you use the `-DcreateStubs=true` flag, it overwrites any existing MBean implementation file.

   > **Note**
   >
   > As of version 9.0 of WebLogic Server, you can also provide a directory that contains multiple MDFs by using the `-DMDFDIR <MDF directory name>` option. In prior versions of WebLogic Server, the WebLogic MBeanMaker processed only one MDF at a time. Therefore, you had to repeat this process if you had multiple MDFs for a given security provider type (for example, multiple authentication providers).

3. Proceed to Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF).

Optional SSPI MBeans or Custom Operations

This section applies to all custom security provider types.

If the MDF for your custom security provider does implement some optional SSPI MBeans *or* does include custom operations, consider the following:

Are you creating an MBean type for the first time? If so, follow these steps:

1. Create a new DOS shell.
2. Type the following command:

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   In the preceding command:

   - The `-DMDF` flag indicates that the WebLogic MBeanMaker should translate the MDF into code.
   - *xmlFile* represents the XML MBean description file (MDF).
   - *filesdir* represents the location where the WebLogic MBeanMaker places the intermediate files for the MBean type.

   Whenever *xmlfile* is provided, a new set of output files is generated.

   Each time you use the `-DcreateStubs=true` flag, it overwrites any existing MBean implementation file.

   > **Note**
   >
   > As of version 9.0 of WebLogic Server, you can also provide a directory that contains multiple MDFs by using the `-DMDFDIR <MDF directory name>` option. In prior versions of WebLogic Server, the WebLogic MBeanMaker processed only one MDF at a time. Therefore, you had to repeat this process if you had multiple MDFs for a given security provider type (for example, multiple authentication providers).

3. If you implemented optional SSPI MBeans in your MDF, follow these steps:
   a. Locate the MBean implementation file.

      The MBean implementation file generated by the WebLogic MBeanMaker is named `MBeanNameImpl.java`. For example, for the MDF named `SampleAuthenticator`, the MBean implementation file to be edited is named `SampleAuthenticatorImpl.java`.

   b. For each optional SSPI MBean that you implemented in your MDF, implement each method. Be sure to also provide implementations for any methods that the optional SSPI MBean inherits.
4. If you included any custom attributes or operations in your MDF, implement the methods using the method stubs.
5. Save the file.
6. Proceed to Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF).

Are you updating an existing MBean type? If so, follow these steps:

1. Copy your existing MBean implementation file to a temporary directory so that your current method implementations are not overwritten by the WebLogic MBeanMaker.
2. Create a new DOS shell.
3. Type the following command:

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   In the preceding command:

   - The `-DMDF` flag indicates that the WebLogic MBeanMaker should translate the MDF into code.
   - *xmlFile* represents the XML MBean description file (MDF).
   - *filesdir* represents the location where the WebLogic MBeanMaker places the intermediate files for the MBean type.

   Whenever *xmlfile* is provided, a new set of output files is generated.

   Each time you use the `-DcreateStubs=true` flag, it overwrites any existing MBean implementation file.

   > **Note**
   >
   > As of version 9.0 of WebLogic Server, you can also provide a directory that contains multiple MDF's by using the `-DMDFDIR <MDF directory name>` option. In prior versions of WebLogic Server, the WebLogic MBeanMaker processed only one MDF at a time. Therefore, you had to repeat this process if you had multiple MDFs for a given security provider type (for example, multiple authentication providers).

4. If you implemented optional SSPI MBeans in your MDF, follow these steps:
   a. Locate and open the MBean implementation file.

      The MBean implementation file generated by the WebLogic MBeanMaker is named `<MBeanName>Impl.java`. For example, for the MDF named `SampleAuthenticator`, the MBean implementation file to be edited is named `SampleAuthenticatorImpl.java`.

   b. Open your existing MBean implementation file (which you saved to a temporary directory in step 1).
   c. Synchronize the existing MBean implementation file with the MBean implementation file generated by the WebLogic MBeanMaker.

      Accomplishing this task may include, but is not limited to: copying the method implementations from your existing MBean implementation file into the newly-generated MBean implementation file (or, alternatively, adding the new methods from the newly-generated MBean implementation file to your existing MBean implementation file); and verifying that any changes to method signatures are reflected in the version of the MBean implementation file that you are going to use (for methods that exist in both MBean implementation files).

   d. If you modified the MDF to implement optional SSPI MBeans that were not in the original MDF, implement each method. Be sure to also provide implementations for any methods that the optional SSPI MBean inherits.
5. If you modified the MDF to include any custom operations that were not in the original MDF, implement the methods using the method stubs.

6. Save the version of the MBean implementation file that is complete (that is, has all methods implemented).
7. Copy this MBean implementation file into the directory where the WebLogic MBeanMaker placed the intermediate files for the MBean type. You specify this as *filesdir* in step 3. (You override the MBean implementation file generated by the WebLogic MBeanMaker as a result of step 3.)
8. Proceed to Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF).

About the Generated MBean Interface File

The **MBean interface file** is the client-side API to the MBean that your run-time class or your MBean implementation uses to obtain configuration data. It is typically used in the initialize method as described in Understand the Purpose of the Provider SSPIs.

Because the WebLogic MBeanMaker generates MBean types from the MDF you created, the generated MBean interface file has the name of the MDF along with the text `MBean` appended to it. For example, the result of running the Simple`SampleAuthenticator` MDF through the WebLogic MBeanMaker yields the MBean interface file `SimpleSampleAuthenticatorMBean.java`.

Use the WebLogic MBeanMaker to Create the MBean JAR File (MJF)

Once your have run your MDF through the WebLogic MBeanMaker to generate your intermediate files, and you have edited the MBean implementation file to supply implementations for the appropriate methods within it, you need to package the MBean files *and the run-time classes* for the custom security provider into an MBean JAR File (MJF). The WebLogic MBeanMaker also automates this process.

To create an MJF for your custom security provider, complete the following steps:

1. Create a new DOS shell.
2. Type the following command:

   ```
   java -DMJF=jarfile -Dfiles=filesdir
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   In the preceding command:

   - The `-DMJF` flag indicates that the WebLogic MBeanMaker should build a JAR file containing the new MBean types.
   - *jarfile* represents the name for the MJF.
   - *filesdir* represents the location where the WebLogic MBeanMaker looks for the files to JAR into the MJF.

   Compilation occurs at this point, so errors are possible. If *jarfile* is provided, and no errors occur, an MJF is created with the specified name.

> **Note**
>
> When you create a JAR file for a custom security provider, a set of XML binding classes and a schema are also generated. You can choose a namespace to associate with that schema. Doing so avoids the possibility that your custom classes conflict with those provided by Oracle. The default for the namespace is `vendor`. You can change this default by passing the `-targetNameSpace` argument to the WebLogicMBeanMaker or the associated WLMBeanMaker ant task.
>
> If you want to update an existing MJF, you must delete the MJF and regenerate it. Do not rename the MJF. The WebLogic MBeanMaker also has a `-DIncludeSource` option, which controls whether source files are included into the resulting MJF. Source files include both the generated source and the MDF itself. The default is false. This option is ignored when `-DMJF` is not used.

The resulting MJF can be installed into your WebLogic Server environment, or distributed to your customers for installation into their WebLogic Server environments.

Install the MBean Type Into the WebLogic Server Environment

To install an MBean type into the WebLogic Server environment, copy the MJF into the *WL_HOME*`\server\lib\mbeantypes` directory, where *WL_HOME* is the top-level WebLogic Server installation directory. This deploys your custom security provider — that is, it makes the custom security provider manageable from the WebLogic Remote Console.

> **Note**
>
> *WL_HOME*`\server\lib\mbeantypes` is the default directory for installing MBean types. (Beginning with WebLogic Server 9.0, security providers can be loaded from `...\domaindir\lib\mbeantypes` as well.) However, if you want WebLogic Server to look for MBean types in additional directories, use the `-Dweblogic.alternateTypesDirectory=`*dir* command-line flag when starting your server, where *dir* is a comma-separated list of directory names. When you use this flag, WebLogic Server always loads MBean types from `WL_HOME\server\lib\mbeantypes` first, then looks in the additional directories and loads all valid archives present in those directories (regardless of their extension).
>
> For example, if `-Dweblogic.alternateTypesDirectory = dirX,dirY,` WebLogic Server first loads MBean types from *WL_HOME*`\server\lib\mbeantypes`, then any valid archives present in `dirX` and `dirY`. If you instruct WebLogic Server to look in additional directories for MBean types and are using the Java Security Manager, you must also update the `weblogic.policy` file to grant appropriate permissions for the MBean type (and thus, the custom security provider). See Using Java Security to Protect WebLogic Resources in *Developing Applications with the WebLogic Security Service*.

You can create instances of the MBean type by configuring your custom security provider using the WebLogic Remote Console, and then use those MBean instances from a graphical user interface, from other Java code, or from APIs. For example, you can use the WebLogic Remote Console to get and set attributes and invoke operations, or you can develop other Java objects that instantiate MBeans and automatically respond to information that the MBeans supply. We recommend that you back up these MBean instances.