BEA AquaLogic Enterprise Security™®

Programming Security For Java Applications

Version: 2.6
Revised: April 2007

Contents

4. Naming Authority

5. Java Security Service Module APIs

6. Developing Applications Using the Java Security Service Module

CHAPTER 1

Introduction and Roadmap

The following sections describe the content and organization of this document:

Scope

This document describes how to implement security in Java applications. It include descriptions of the Security Service Application Programming Interfaces and programming instructions for implementing security in Java applications

Documentation Audience

This document is intended for the following audiences:

- Application Developers—Developers who are Java programmers who focus on developing Java applications, incorporating security into Java applications and Enterprise JavaBeans (EJBs), and who work with other engineering, quality assurance (QA), and database teams to implement security features. Application Developers have in-depth working knowledge of Java (including J2EE components such as servlets/JSPs and JSEE).

- Security Architects—Individuals who are responsible for designing and implementing the overall security architecture for their organization, evaluating BEA AquaLogic Enterprise Security features, and determining how to best implement policies. Security Architects have in-depth knowledge of Java programming, Java security, and network security, as well as knowledge of security systems and leading-edge security technologies and tools.
- Security Developers—Developers (including third-party developers) who focus on defining the system architecture and infrastructure for security products and who develop custom security providers for use with BEA AquaLogic Enterprise Security services. Security Developers work with Security Architects to ensure that the architecture is implemented according to design specifications and that it does not introduce any security holes. Security Developers also work with administrators to ensure that security is properly configured. Security Developers have a solid understanding of certain concepts, including authentication, authorization, and auditing, and an in-depth knowledge of Java and security provider functionality.

Guide to this Document

This document is organized as follows:

- Chapter 2, "Introduction," introduces the Java Security Service Module product and describes its components.
- Chapter 3, "Java Security Service Module Concepts,"describes Java Security Service Module concepts as they relate to BEA AquaLogic Enterprise Security.
- Chapter 4, "Naming Authority," describes naming authorities in the context of the Java Security Service Module.
- Chapter 5, "Java Security Service Module APIs," describes the APIs that you use to develop Java applications using the Java Security Service Module.
- Chapter 6, "Developing Applications Using the Java Security Service Module," provides step-by-step procedures for developing Java applications. The procedures include code fragments that demonstrate how to implement each programming step.

Related Information

The BEA corporate web site provides all documentation for BEA AquaLogic Enterprise Security. Other BEA AquaLogic Enterprise Security documents that may be of interest to the reader include:

- *WSDL Documentation for the Web Service Interfaces*—This document provides reference documentation for the Web Services Interfaces that are provided with and supported by this release of BEA AquaLogic Enterprise Security.
- *Administration and Deployment Guide*—This document provides step-by-step instructions for performing various administrative tasks.
- *Integrating ALES with Application Environments*—This document describes important tasks associated with integrating AquaLogic Enterprise Security into application environments.
- *Policy Managers Guide*—This document how to write access control policies for BEA AquaLogic Enterprise Security, and describes how to import and export policy data.
- *Installing Security Services Modules*—This document describes how to install ALES Security Services Modules, including the Web Services Security Service Module.
- *Developing Security Providers* —This document provides security vendors and security and application developers with the information needed to develop custom security providers.
- *Javadocs for Security Service Provider Interfaces*—This document provides reference documentation for the Security Service Provider Interfaces that are provided with and supported by this release of BEA AquaLogic Enterprise Security.
- *Javadocs for Java API*—This document provides reference documentation for the Java Application Programming Interfaces that are provided with and supported by this release of BEA AquaLogic Enterprise Security.

CHAPTER 2

Introduction

The Java Security Service Module is a java-based product that allows an application developer to access sets of interfaces to define and implement security related information and requirements specific to a Java application. These interfaces support the most commonly required security functions and are organized into services that are logically grouped by functionality.

After you use the Java Security Service Module interfaces to implement security functions in your Java application, you can deploy and run your application on any instance of a Java Security Service Module runtime that supports the configuration requirements of your application.

The Java Security Service Module offers five security services: Authentication Service, Authorization Service, Auditing Service, Role Service, and Credential Mapping Service. The name of each service indicates the type of function it is used to implement within a Java application. Each of these services is discussed in more detail later.

Because most major functions required by Java applications developed using the Java Security Service Module are performed within the security framework, this architecture has several benefits, including:

- **Application uptime is maximized**—Because the Java Security Service Module runs in-process with your application, your application is tightly coupled with its security system. If the application is running, the security system is running as well. Even if providers are designed to use external resources (such as an LDAP server), the Security Service Module is designed to continue operating and to fail securely if that remote resource becomes unavailable for any reason.
- **On-the-wire security is enhanced**—Because security information is provisioned into the Security Service Module on the same machine as the Java application, the Security Service

Module does not need to fetch this information from the network. This prevents unscrupulous programs from listening in on these security-related conversations to determine vulnerability.

- **Performance is improved**—Another aspect of having security and policy data loaded locally is that the Security Service Module can more quickly retrieve policy information, resulting in improved performance. The locality of security metadata combined with the in-process nature of the Security Service Module allow for a very fast evaluation of security questions.

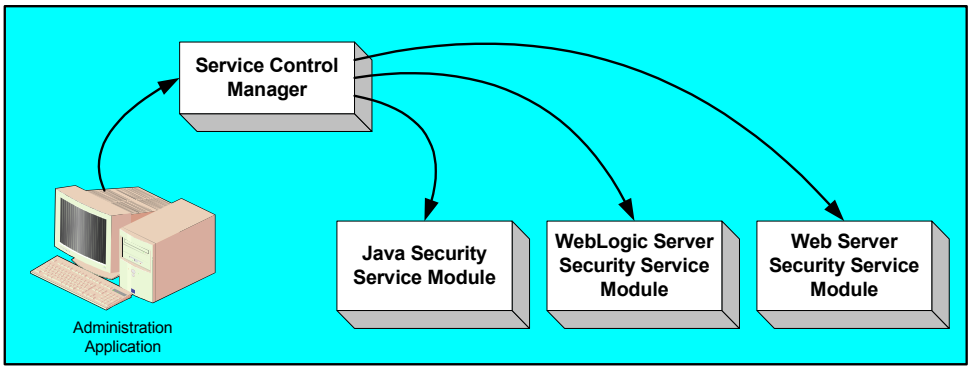
The following topics provide more information on the Java Security Service Module:

- "Java Security Service Module Environment" on page 2-2
- "Java Security Service Module Functional Description" on page 2-4

Java Security Service Module Environment

Figure 2-1 shows the major components that make up the Security Service Module environment.

Figure 2-1 BEA AquaLogic Enterprise Security Service Modules



- Administration Application

  The Administration Application allows you to manage and configure multiple Security Service Modules. While Security Service Modules specify and consume configuration data and then services security requests accordingly, the Administration Application allows you to display the security providers that are plugged into the security framework and to display and modify the configuration data for those providers.

- Service Control Manager

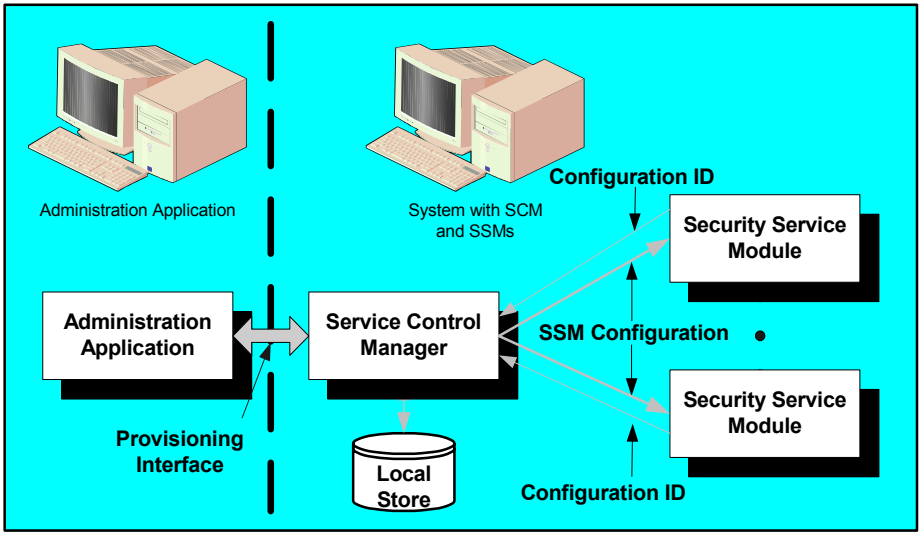
  The Service Control Manager (SCM) is an essential component of the BEA AquaLogic Enterprise Security configuration provisioning mechanism and of a fully-distributed security enforcement architecture.

  Note: AquaLogic Enterprise Security version 2.5 removed the requirement that a Service Control Module (SCM) be installed on each system where one or more Security Service Modules (SSMs) are installed.

  An Service Control Manager is a machine agent that exposes a provisioning interface to the Administration Application to facilitate the management of a potentially large number of distributed Security Service Modules. A Service Control Manager can receive and store metadata updates, both full and incremental, initiated by the Administration Application.

  The Administration Application uses the provisioning mechanism of the Service Control Manager to distribute configuration and policy data to each Security Service Module where it is consumed locally (see Figure 2-2). Security Service Modules (which can be distributed throughout an enterprise) can be embedded in Java applications, application servers, and web servers. After you use the Administration Application to configure an instance of an Security Service Module with configuration and policy data, the Security Service Module does not require any additional communication with the Service Control Manager to perform security functions. However, the Service Control Manager maintains communication with the Security Service Module to distribute full and incremental updates.

Figure 2-2 Deploying Configuration and Policy Data to Security Service Modules



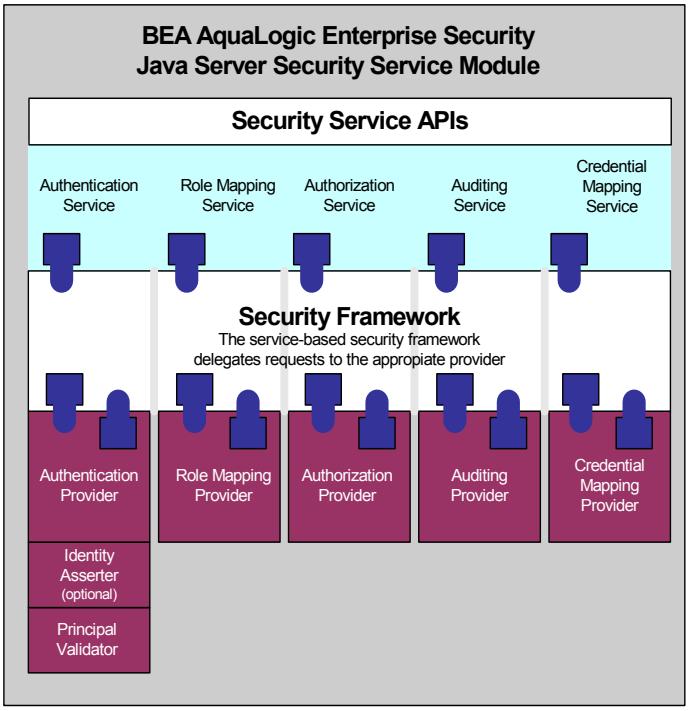
Java Security Service Module Functional Description

Figure 2-3 shows the major components of the Java Security Service Module. The Java Security Service Module comprises the security service APIs, the security framework, and the security providers that you configure in any given instance of a Java Security Service Module runtime.

The following topics describe these components:

- "Security Service APIs" on page 2-5.
- "Security Framework" on page 2-6
- "Security Providers" on page 2-7

Figure 2-3 BEA AquaLogic Enterprise Security Java Security Service Module Components



Security Service APIs

The Java Security Service Module supports the following security service APIs:

- "Authentication Service" on page 2-6.
- "Authorization Service" on page 2-6.
- "Auditing Service" on page 2-6.
- "Role Service" on page 2-6.
- "Credential Mapping Service" on page 2-6.

Authentication Service

The Authentication Service provides functions to an application related to establishing, verifying, and transferring a person or a process. Thus, the Authentication Service provides two main functions: authentication and identity assertion.

Authorization Service

The Authorization Service is a service that allows an application to determine if a specific identity is permitted to access a specific resource. This decision may then be enforced in the application directly at the policy enforcement point.

Auditing Service

The Auditing Service allows an application to log events based upon activity related to enterprise security. The Java Security Service Module runtime uses these mechanisms to log appropriate data when events occur.

Role Service

The Role Service allows an application to extract role information about specific identities and resources within the context of the application. These roles may then be used to customize interfaces.

Note: Roles themselves should not be used for authorization, as many policies allowing or disallowing access to a resource may be written against a role. It is best that you use the Authorization Service to determine actual rights.

Credential Mapping Service

The Credential Mapping Service allows an application to fetch credentials of certain types that are associated with a specific identity for a specific resource. These credentials may then be used on behalf of that identity to provide some privileged function, such as logging into a database or sending e-mail.

Security Framework

The primary function of the Security Framework is to provide an application programming interface (API) that security and application developers use to implement security functions in Java applications. Within that context, the Security Framework also acts as an intermediary between security functions that you implement in Java applications using the Java security service APIs and security providers configured into the Java Security Service Module. For more information on the Security Framework, see *Introduction to BEA AquaLogic Enterprise Security*.

Security Providers

When you install the Java Security Service Module, a JAR file is deployed that contains all the default security providers that ship with the product. However, before any of the security providers can be used, you must use the Administration Application to configure them in the Java Security Service Module. You have the option of configuring either the default security providers that ship with the product or custom security providers, which you develop or purchase from third-party security vendors. The Java Security Service Module supports the following types of security providers:

- Authentication provider
- Authorization provider
- Auditing provider
- Credential mapping provider
- Identity asserter
- Principal validator
- Role mapping provider

For more information on the security providers, see *Introduction to BEA AquaLogic Enterprise Security*. For information on developing custom security providers, see *Developing Security Providers for BEA AquaLogic Enterprise Security*.

CHAPTER 3

Java Security Service Module Concepts

This section describes the concepts that you need to understand to use the Java Security Service Module product effectively. The concepts are grouped into two categories:

"General Concepts" on page 3-1

"Concepts that Relate to Interfaces and Classes" on page 3-3

General Concepts

The following sections describe the general concepts:

- Anonymous User
- Compatibility
- DeepTokenEnumeration
- TokenEnumeration

Anonymous User

An anonymous user is a special user that includes everyone that is not authenticated. You use the `com.bea.security.SecurityRuntime.getAnonymousIdentity()` method to get the anonymous user `AuthenticIdentity` from the security Runtime object (see "RuntimeAction" on page 3-7).

Compatibility

Compatibility relates to versions of the services required by the Java Security Service Module application and versions of the service APIs in the Java Security Service Module. If the versions are not compatible, the portion of the application that uses that service API will not work and security holes may exist in the application. Therefore, you may choose to check each service API for compatibility before you attempt to use it.

All the service APIs support an `isCompatible()` method which may be used by the application to determine whether the version of the service API is compatible with the application.

DeepTokenEnumeration

A deep-token enumeration recurses into references to other naming authorities until the given name is completely resolved into a token. A token, in this case, is analogous to an attribute.

As with `TokenEnumeration`, `DeepTokenEnumeration` is another area where an application developer may choose to override the default implementation for a given naming authority. If you want recursion based on the value of a given attribute, then you can implement this extraneous logic in a custom naming authority class.

TokenEnumeration

One duty of a naming authority is to resolve names into a series of tokens. A token, in this case, is analogous to an attribute. Standard token evaluation parses a name and returns a list of `NameAttributeValues` that contains the value and the `NameAttributeType`. If that attribute references another authority, this reference is not resolved during this type of enumeration.

Token enumeration may be one area in particular where an application developer may choose to override the base class. If a name is sufficiently complex, there may not be a straightforward way for a naming authority to parse it into its attribute forms by the built-in parsing rules.

Concepts that Relate to Interfaces and Classes

The following concepts relate to the Java application programming interface (API) interfaces and classes:

- AccessResult
- Application Configuration
- Application Context
- AppContextElement
- AttributeValueEnumeration
- AuthenticIdentity
- ContextAuditRecord
- HashMapContext
- IdentityRole
- NameAttributeType
- NameAttributeValue
- NamedObjects
- NamingAuthority
- NamingAuthorityManager
- PolicyDomain
- RuntimeAction
- RuntimeResource
- SecurityRuntime
- ServiceType
- ServiceVersion
- SimpleContextElement

AccessResult

An `AccessResult` is the object returned to an Java Security Service Module application after an access decision is made. Primarily, this object returns a Boolean value as to whether access is permitted (`true`) or denied (`false`) through its `isAllowed()` method. The `AccessResult` object also returns a date and time as to when the decision was evaluated and optionally lets an application re-query the decision without having to locate the original arguments.

Application Configuration

The application configuration is a representation of your application that you pass into the Java Security Service Module when you initialize it. The application configuration is passed in as the `AppConfig` object. The `AppConfig` object is used to configure a runtime based on the configuration of your application. The `AppConfig` object supports a method for adding an XML file that contains naming authority definitions to the configuration object. On initialization of the Java Security Service Module, this XML file is loaded into the Naming Authority Manager.

Application Context

The application context is a description of the environment as it relates to your application. It is a collection of names and values that the security providers can query to get information from the application. Thus, the application context enables the providers to react appropriately and to be configured properly. Specifying the application context also allows the application developer to publish name/value pairs from within the application to the Java Security Service Module runtime. The Java Security Service Module may use these values to assist it in providing security decisions or information.

Note: `The com.bea.security.HashMapContext` class provides a sample implementation of an `AppContext` interface. As a developer, you have the option of using the `HashMapContext` class within an application as the specific way to implement an `AppContext` object or using the `com.bea.security.AppContext` interface to implement your own version.

AppContextElement

The `AppContextElement` interface is a simple interface for representing a name/value pair. This interface is applied to application specific context entries so that those values can be evaluated as context to a security decision. `The SimpleContextElement` class provides a sample implementation of an `AppContextelement`.

AttributeValueEnumeration

The `AttributeValueEnumeration` class is used to implement naming authorities. The `AttributeValueEnumeration` class is a representation of an ordered list of name attribute values. This class is used to return an ordered list for named objects. It provides an enumeration interface to the ordered list as well as several methods to return different views of that list. This class is used and consumed by named objects for constructing string names.

AuditRecord

The `AuditRecord` class allows a user to log an audit message conforming to a naming convention.

AuthenticIdentity

The `AuthenticIdentity` class represents an authenticated person or process. There is also a special case of `AuthenticIdentity` that represents the anonymous, or non-authenticated user. This class is meant to be an opaque representation of the user. The `toString()` method provides a human readable string that represents this identity suitable to display or for logging.

ContextAuditRecord

The `ContextAuditRecord` class allows a user to log an event or audit message that conforms to a naming convention and automatically extracts elements of the event from the application context of the event.

HashMapContext

The `HashMapContext` class is a simple implementation of an application context based on top of a java hashmap. Applications may choose to use this class as a basis for their application context or implement their own.

IdentityRole

An `IdentityRole` is any representation of a canonical role assigned to an identity for a particular resource, action, and application context. An `IdentityRole` contains a role name and a description. The `IdentityRole` class represents a role assigned to an identity. You use the `com.bea.security.RoleService.getrole()` method to get the appropriate set of roles assigned to an identity. For example, an `IdentityRole` for `admin` includes a description that says administrators can do this or that. An `IdentityRole` is read only.

NameAttributeType

The `NameAttributeType` class is used to implement naming authorities. The `NameAttributeType` class represents a single named field within a name as defined by a naming authority.

NameAttributeValue

The `NameAttributeValue` class is used to implement naming authorities. The `NameAttributeValue` class subclasses `NameAttributeType` adding a value in addition to the type. `NameAttributeType` is the class used when a name is tokenized by its naming authority.

NamedObjects

The `NamedObject` class is used to implement naming authorities. The `NamedObject` is the base class for all objects managed by a naming authority. It provides accessor functions for consumers of name managed objects and core fields that reference which naming authority manages this object and whether or not this object is validated. In the BEA AquaLogic Enterprise Security environment, a named object is any object that conforms to a naming code, structure, or convention. There are four types named objects: `AuditRecord`, `ContextAuditRecord`, `RuntimeAction`, and `RuntimeResource`.

You define the structure of named objects when you design and implement the naming authority that your Java Security Service Module application uses. Because the named objects conform to the naming conventions and structures that you specify in the naming authority, the BEA AquaLogic Enterprise security providers can consult the naming authority and determine what values are contained in the object. Thus, named objects along with naming authorities provide a data-driven way for your application to control which named objects are supported and to define the structure of and the values used by those objects. This capability, in effect, removes all limitations on the naming and structure of objects that can be supported by applications developed using the Java Security Service Module APIs and the Security Services Provider Interfaces (SSPI).

NamingAuthority

The `NamingAuthority` class is used to implement naming authorities. The `NamingAuthority` class is the base class for all instances of naming authorities. If an application programmer wants to implement a naming authority in Java to streamline name parsing, this class would the base class. Default implementations of the `getTokenEnumeration` and `getDeepTokenEnumeration` functions are provided in this base class as a convenience to application developers.

NamingAuthorityManager

The `NameAuthorityManager` class is used to implement naming authorities. The `NameAuthorityManager` class is a central registrar for all naming authorities that the Java Security Service Module recognizes. Within an Java Security Service Module if the name is not registered with its naming authority's manager, then the name effectively does not exist.

This class is also responsible for managing the dependencies of the naming authorities. If a naming authority's dependencies are not met, then it will not be available to the Java Security Service Module runtime. The naming authority remains associated with the Naming Authority Manager, and every time a new authority is added, it's dependencies are re-evaluated and enabled once the dependencies are all met. This class is also responsible for loading XML definitions of authorities and validating them.

PolicyDomain

The `PolicyDomain` class is used to implement naming authorities. The `PolicyDomain` object provides security services to an Java Security Service Module application that is scoped within a single policy domain. This object allows an application to determine which services are available and create instances of services.

RuntimeAction

The `RuntimeAction` object represents an action or privilege to the Java Security Service Module runtime. `RuntimeAction` is a `NamedObject` and, therefore, is managed by a naming authority that determines its format and delimiters.

RuntimeResource

The `RuntimeResource` object represents a resource to the Java Security Service Module runtime. A `RuntimeResource` is a `NamedObject` and, therefore, is managed by a naming authority that determines its format and delimiters. A `RuntimeResource` also contains functions that allow it to return a reference to its parent or a sibling (as determined by a string suffix) or a child (as determined by a string suffix). Each parent reference is managed by the naming authority as the original resource.

SecurityRuntime

The `SecurityRuntime` object is the largest abstract concept within an Java Security Service Module application. The `SecurityRuntime` class provides a way for the Java Security Service Module application to query the runtime for its capabilities and to fetch several other application level resources. Specifically, `SecurityRuntime` objects are used to initialize, instantiate, and fetch `PolicyDomain` objects for use within an application, and to fetch the instance of the Naming Authority Manager. The Java Security Service Module runtime provides your application environment. There can only be one runtime of a certain type in a specific application environment.

An application usually has one policy domain, although container applications can have more than one. In the Java Security Service Module API, a `PolicyDomain` object aggregates services related to that policy domain into one interface. To get an instance of a `PolicyDomain` object, a Java Security Service Module application developer writes code to initialize and fetch an instance of the `SecurityRuntime` object (a singleton). Once this instance is fetched, the application can query the runtime to ensure that a policy domain exists, and then create an instance of a policy domain.

ServiceType

`ServiceType` indicates what type of service is available. The `com.bea.security.ServiceType` class represents a service that is offered for use by the application and the Security Service Module runtime. Five service types are supported by the `ServiceType` class to facilitate their use in your applications: Audit Service, Authentication Service, Authorization Service, Credential Mapper Service, and Role Service. However, you may expand the service types beyond the knowledge of the `ServiceType` class and used them with Java Security Service Module applications.

ServiceVersion

The `ServiceVersion` is an encapsulation of the current version of the service and is useful for managing compatibility within your application. The `com.bea.security.ServiceVersion` class is used to represent a service version. A service version contains a major number, minor number, and a string for the patch number (which is represented as a string), for example 1.0sp1, or 1.1sp2, etc.

When the service type is returned by the service, the application can determine which version of the service is running on the instance of a Java Security Service Module runtime in use. The service version can then be used by the application to determine whether the application is compatible with the service that is running. For example, if you have programmed your application to be compatible with Authentication Service version 1.0, then you can instantiate an Authentication Service version 1.0, pass it into the Authentication Service using the `com.bea.security.AuthenticationService.isCompatible` method. The `isCompatible` method returns `COMPATIBLE`, `NOT_COMPATIBLE`, `COMPATIBLE_DEPRICATED`, or `COMPATIBLE_UNKNOWN` to indicate whether or not the Authentication Service API that you just fetched is compatible with API version 1.0.

SimpleContextElement

The `SimpleContextElement` class is used with `HashMapContext` for Java Security Service Module applications that want to present their context information using these simple classes.

CHAPTER 4

Naming Authority

A naming authority has two functions. First, it provides a scope for a name, which ensures that the name is unique within that scope. Second, it defines the format of the name, breaking the name into named fields separated by delimiters.

The following topics provide information to help you understand naming authorities and how they are used in Java Security Service Module (SSM) applications:

Why Use a Naming Authority?

Naming something uniquely has always been an issue in software. As the number of objects grows within a software system, the need for unique names increases and the problems of managing that uniqueness become more difficult.

Some systems rely on a certain amount of randomness to create unique identifiers. These systems work well with distributed environments as they are based upon statistical models that allow multiple authors to generate identifiers randomly with the same effect as a single governing entity. The short-coming of this approach is that the statistical model permits duplication as the number of objects in the system grows—it only guarantees that some miniscule percentage of objects will have duplicate IDs. In addition, since the IDs are random, they do not contain an inherent structure, are difficult to read by humans, and are often difficult to optimize for query and retrieval.

A common system for generating unique identifiers (IDs) uses existing knowledge of uniqueness in the form of a system network ID or CPU serial number, in addition to other elements. This limits the requirements of the uniqueness within one computer system, that is, if an ID is unique within one system, then by appending a unique system ID to it, it can be unique within any distributed-system environment. While this reduces the scope of the problem in that it eliminates the possibility of duplication, these identifiers are, by nature, just as random as entropy driven mechanisms and are equally difficult to read or optimize.

Other systems employ an object registry model, where a central registry issues IDs to all software systems. The problem with using a central registry is that it can be quickly overwhelmed. To deal with this problem, systems employing the object registry model often segment their ID pool into several sub-registries which are then handled by multiple servers. In the case of numeric IDs, IDs 1-10000 might get allocated to server1, IDs 10001 to 20000 to server2, etc. This model, although having some similarity to a naming authority structure, still requires a central, master registry to determine what blocks of IDs are given to the helper systems, and so it is just a modified version of a central registry.

The Java Security Service Module Naming Authority provides solution to all of the problems associated with naming systems discussed thus far. The naming authority model employs the segmentation of responsibility present in a registry-based system, but expands it so that IDs become readable by humans, are hierarchical, and are easily optimized. Furthermore, cooperating naming authorities are independent of each other and are only responsible for guaranteeing uniqueness within themselves. Thus, using multiple naming authorities eliminates the performance problems inherent in systems that use a central registry. It also allows multiple

systems to use the same names, because the naming authority model guarantees uniqueness by using different naming authorities.

How the Use of Naming Authorities Guarantees Uniqueness

Could “Rose” from the nursing home be different from “Rose” from the flower shop? Such is the problem with names; the same ones are used to mean very different things. The Java Security Service Module Naming Authority remedies this problem.

Any name in the real world has an implied authority already. In the example above with the name “Rose,” one authority is the residence list of the nursing home and another authority would is the inventory of the flower shop. Only in relating the name AND the authority can a person properly communicate the uniqueness of the name “Rose.”

Name Scoping

The scoping functions of the name authority allow identical names to be recognized as unique. For example, OrderEntryForm may exist in both a fulfillment application and a point of sales (POS) application. The scope for a name can be recognized by associating the name OrderEntryForm with the naming authority FULLFILLMENT or the naming authority POS.

How a Naming Authority Adds Structure

A name can contain more information than mere uniqueness. It can also have structure. For example, the name for a woman named Rose who works in a nursing home in the USA might include her first name and last name, Rose Bud. But in other parts of the world, her name may include both her maiden name and married name in addition to her first name, Rose Pedalsweet Bud. Therefore, the name “Rose Bud” in the USA has a different structure than “Rose Pedalsweet Bud” as it may be presented in other cultures. Within the authority of a specific nursing home, either name is sufficient to identify Rose as a unique person. By having knowledge of whether the nursing home is in the United States of America (USA) or in another place, such as the United Kingdom (UK), you can gain knowledge of structure of the name. For example, the naming authority for the residence list of the USA nursing home could be name USAResidenceList, and the residence list of a UK nursing home could be UKresidenceList.

Name Parsing and Formatting

A name may define fields within itself. For example a form may contain fields, fields may contain functions, and so on. The name might be defined as having FORMNAME, FIELDNAME, and FUNCTION fields with a delimiter of "." (period). This would allow a name like

"OrderEntryForm.name.setText" to then be parsed into three name/value pairs "FORMNAME" = "OrderEntryForm", "FIELDNAME" = "name", "FUNCTION" = "setText". By combining naming authorities, sub-naming authorities, and delimiters into a parse tree, a very flexible way of defining names can be formed.

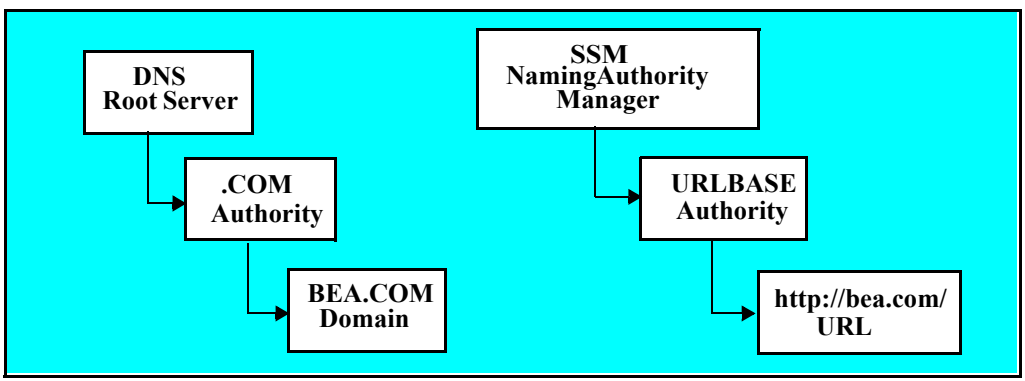
The NamingAuthority also allows an object to be represented in its string form that follows the NamingAuthority definition. If through the use of naming authorities and sub-authorities a name cannot be represented, a custom class can be created to provide custom or optimized parsing.

Associating a Named Object with a Naming Authority

If a naming authority is responsible for differentiating between names within a distributed system environment, there needs to be a way to indicate that one name is associated with a particular authority versus any other. Within a software system, this is easily done by providing a reference for a named object to its authority.

Outside of a software environment, a tree-based naming convention is often employed to enforce uniqueness and provide a visual separation of names within a system. One system that uses this method is the Internet domain name system (DNS). The domain name system is based upon some theoretical "root server" in which defines naming authorities, domains, sub-domains, and finally names. The root server provides a starting place for searching for naming authorities. In the Java Security Service Module, the Naming Authority Manager fills a similar role.

Figure 4-1 Comparison of DNS with a Java Security Service Module Naming Authority



By querying the root servers, you may discover that there are many naming authorities present. Some naming authorities that we are all familiar with are ".com", ".org", ".edu", ".us", ".biz".

Just as these authorities are registered with the root server, Internet domains must be registered with the proper authority before they can be recognized as valid domains. The Naming Authority Manager serves to register naming authorities in the same way. Before a name of type "URLBASE" can be recognized as a valid name, the naming authority for "URLBASE" must be registered with the Naming Authority Manager.

Types of Authorities Supported

The Java Security Service Module naming authority supports the following types of authorities:

- "Sub-Authorities" on page 4-5
- "Peer Authorities" on page 4-6
- "Self-Referencing Authorities" on page 4-6

Sub-Authorities

Just as DNS naming systems can have sub-domains, the Java Security Service Module naming authority can have sub-authorities. A sub-authority is useful when a name contains one or more other formatted names within it. In the case of a URL, which is formatted as:

```
protocol://servername:port/path/filename?arg1=val&arg2=val
```

where:

- `protocol` is either HTTP or HTTPS
- `servername:port` is the name and port number of the system running the application. For example, the servername might be `cracker` and the port is typically `7001` for HTTP connections and `7002` for HTTPS connections.
- `path` is the directory on the server, such as `/ales/docs21`.
- `filename` is the page being served, such as `index.html`.
- The `args` are query_string arguments, such as `?SectionID=26&SubSectionID=398&ArticleID`.

A URL contains a `filename` portion that may be represented in the `URL:FILENAME` format:

```
basename.extension
```

By linking the `filename` part of the URLBASE naming authority to a `URLBASE:FILENAME` sub-authority, the `filename` can be further broken down into a basename and extension. Note that the sub-authority name `URLBASE:FILENAME` does not have to include the prefix `URLBASE:`,

but by including the name of the parent as a prefix it can be distinguished as different from a generic `FILENAME` authority. This naming convention is described by the specific instance of the naming authority.

Peer Authorities

In addition to sub-authorities, a Java Security Service Module naming authority can also contain references to peer authorities. A peer authority is more general in nature and exists in another authority namespace (such as at the top level). Using the same URL format described in the previous section, the `servername` portion might be represented by a DNS naming authority as:

```
nodename.subdomain.domain.authority
```

This effectively changes the URL format from:

```
protocol://servername:port/path/filename
```

to:

```
protocol://nodename.subdomain.domain.authority:port/path/filename
```

By referencing this general DNS naming authority (which may be referenced by many other naming authorities), your authority holds a dependency on this authority and may not function well without it. In the case of the URL, if a DNS naming authority were not present, the servername would merely be represented as `servername=www.bea.com` and so the naming authority would continue to function in a reasonable manner. If a naming authority absolutely requires another authority, then it must list the other authority explicitly as a dependency.

Sub-authorities, by their nature, inherit all the dependencies of their parent, plus their parent, in addition to any dependencies to peer authorities that may exist.

Self-Referencing Authorities

An authority can also be configured to reference itself in a recursive manner. For example, if a list of an unknown number of arguments is present, an authority that represents the name of one argument can extract the first argument, represent it, and then reference itself for the remaining arguments.

Supported Naming Authority Attributes

Six different naming attribute types can be used with naming authorities.

- `SINGLE_VALUE_PREFIX`
- `MULTI_VALUE_PREFIX`

- SINGLE_VALUE
- MULTI_TOKEN
- SINGLE_VALUE_TERMINAL
- MULTI_VALUE_TERMINAL

Notice that you can group the attribute types into two categories: single and multiple.

- Single-type attributes have one name and one value and are the simpler of the two types.
- Multi-type attributes have one name but can contain many values. Additionally, Multi-type attributes may also reference another naming authority to indicate that the values contained within it are governed by another naming convention.

Attribute types have specific behaviors according to their type. The following topics describe the behaviors of each type:

- SINGLE_VALUE and MULTI_VALUE PREFIX Attribute
- SINGLE_VALUE and MULTI_VALUE TERMINAL Attribute
- SINGLE_VALUE Attribute
- MULTI_TOKEN Attribute

SINGLE_VALUE and MULTI_VALUE PREFIX Attribute

All prefix attributes (both single and multiple) must exist at the beginning of the name. The naming authority is considered invalid if it attempts to define prefix attributes anywhere else in the attribute list. However, you can use as many prefix attributes as are necessary in sequence at the start of an attribute list.

A prefix attribute provides a means for any name to be separated into attributes. The first parts are prefix attributes; whatever is left is then mapped to other attribute types. Consequently, single-value prefix attributes are the highest priority attributes in an attribute list.

Multi-value prefix attributes provide the same functionality as single-value prefix attributes except that they can represent more than one value. By associating this attribute with another naming authority, which may use another delimiter, this attribute can be decomposed further into other name/value pairs. As with single-value prefix attributes, multi-value prefix attributes are of the highest priority within a naming authority.

SINGLE_VALUE and MULTI_VALUE TERMINAL Attribute

A terminal attribute (both single and multiple) is meant to indicate that a section of the name exists at the end of a name. Just as prefix attributes must exist at the beginning, terminal attributes must be at the end of a name definition. Many terminal attributes can be used when defining a name.

Terminal prefixes are the second most important attribute within a naming system. Once the prefix attributes are determined, the terminal attributes are evaluated and whatever is left is mapped to the remaining attribute types.

A `MULTI_VALUE_TERMINAL` attribute provides a mechanism where a single terminal attribute type can contain many values. In addition, it can also reference another authority for further evaluation.

SINGLE_VALUE Attribute

A single-value attribute can exist anywhere and is a single name/value pair. This attribute is the simplest and most generic.

MULTI_TOKEN Attribute

A multi-token attribute is a "catch all" for all the sections of the name that do not map to any other attribute by their placement. Consequently, a naming authority may only have one multi-token attribute defined or else it is considered invalid. Multi-token attributes join the remaining parts of the name together, using the naming authority primary delimiter (the first one in its delimiter list), into one value delimited by that delimiter. If a multi-token attribute references another naming authority, then instead of using its own primary delimiter, the attribute uses the primary delimiter of the referenced authority during a shallow token enumeration. For more information, see "Registering a New Naming Authority" on page 6-7.

Setting up ALES Naming Authorities

You can use either the Java API or XML definition files to define naming authorities. See "Defining and Implementing a Naming Authority" on page 6-5 for complete information.

Java application developers can use the Java API to streamline name parsing. The `com.bea.security.NamingAuthority` class provides the base class for the naming authority implementation. Default implementations of the `getTokenEnumeration` and

`getDeepTokenEnumeration` methods are provided in this base class as a convenience to application developers.

Naming authorities can be defined using XML. During parsing of the XML definitions, the following processing occurs:

- Elements that have names of "AUTHORITYDEFINITION" (case insensitive):
  - If the element has an attribute of "class" then that class is instantiated and used as the naming authority.
  - If the "class" attribute is not present, then attributes of "name" and "delimiters" are looked up. If these attributes exist then a DataDrivenAuthority (com.bea.security.DataDrivenAuthority) is created and used as the naming authority. The delimiters are used to tokenize the name according to the naming authority's rules.
- Elements that have names of "AUTHORITY" (case insensitive):
  - Used to add a dependency on the authority within which this Authority XML element exists.
- Elements that have names of "ATTRIBUTE" (case insensitive):
  - Name, type, and authority attributes are found out, and a new `com.bea.security.NameAttributeType` is created using that information and it is added as an attribute to the current Authority entity.

Pre-Configured Naming Authorities

A library of pre-configured naming authorities is provided with Java Security Service Module, for example, the URLBASE naming authority. However, these naming authorities should not be used without modification. Instead you can reference the pre-configured authority as the sole sub-authority of your own custom naming authority. For example, if the `BEA_URLBASE` naming authority references the pre-configured URLBASE naming authority, `BEA_URLBASE` inherits all of the fields and rules of the URLBASE authority but still has a unique identity of `BEA_URLBASE` allowing you to configure other `URL` types such as `PARTNER_URLBASE` and `COMPETITOR_URLBASE`. With this approach, you do not have to design your own URL naming authority and you can still have your own unique authority. On the other hand, if everyone were to use the URLBASE naming authority as is, it would defeat the purpose of using a naming authority, because there would be no way to differentiate between a BEA named object of `URLBASE` type and a Sun named object of `URLBASE` type.

The sections that follow describe the pre-configured naming authorities.

URLBASE

The URLBASE naming authority uses the forward and backward slash characters (/) and (\) as the delimiters. This authority can be used to tokenize a URL type of name. Typically, the URL name is expected to have tokens such as protocol, server name, port, path and file specification. For example, the URLBASE naming authority will be able to handle a name such as http://localhost:2010/testarea/testpage.html. This name will be tokenized into name value pairs such as protocol = http:, serverandport = localhost:2010, path = testarea, and filespec = testpage.html.

A more comprehensive example of the name handled by this naming authority is - http://admin:secret@www.mycompany.com:8888/one/two/three/four/index.html?question=what&answer=test.

This name can be shallow-parsed into name value pairs, as follows:

- protocol = http:
- usernamepasswd = admin:secret
- serverandport = www.mycompany.com:8888
- path = one/two/three/four
- filespec = index.html
- namevaluepairs = question=what&answer=test

UNC

The UNC naming authority uses the forward and backward slash characters (\) and (/) as delimiters. This authority can be used to tokenize UNC path type of names. Typically, the UNC path is expected to have tokens such as server name, share name, path and file name. For example, the UNC naming authority will be able to handle a name such as \\myserv\myshare\mydir\myExec.exe. This name will be tokenized into name value pairs such as servername = myserv, sharename = myshare, path = mydir, and filename = myExec.exe.

ARME_RESOURCE_AUTHORITY

The ARME_RESOURCE_AUTHORITY naming authority uses the forward slash character (/) as a delimiter. The name handled by this authority is expected to have two parts: app and object. An

example name parsed by this naming authority is `ASI/test`, which will be parsed into name value pairs of `app = ASI` and `object = test`.

Example of Using ARME_RESOURCE_AUTHORITY From the Java API

```
String resource = "ASI/test";

String actionStr = "GET";

RuntimeResource RTResource = new RuntimeResource(resource,
"ARME_RESOURCE_AUTHORITY");

RuntimeAction RTAction = new RuntimeAction(actionStr,
"ARME_ACTION_AUTHORITY");

AccessResult accessResult = tryAuthorize(authService, authIdentity,
RTResource, RTAction, appContext);
```

Action Naming Authorities

This section describes the action naming authorities.

ARME_ACTION_AUTHORITY

The `ARME_ACTION_AUTHORITY` naming authority uses the forward slash character (`/`) as a delimiter. Typically, a name is expected to have a single token with a name of `privilege`. Examples of the name handled by this naming authority are `GET`, `READ`, and `WRITE`.

SIMPLE_ACTION

The `SIMPLE_ACTION` naming authority uses the period character (.) as the delimiter character. Typically, a name is expected to have several action tokens. Examples of names parsed by this authority are `PUT`, `GET`, `view`, and so forth.

For a name like `VIEW.refresh`, the name value pairs will be `action = VIEW` and `moreaction = refresh`.

Audit Event Naming Authorities

This section describes the audit naming authorities.

AUDITBASE

The `AUDITBASE` naming authority uses the forward slash character (`/`) as the delimiter. A typical `AUDITBASE` name is expected to have type, severity, and message tokens. For example, the `AUDITBASE` naming authority will be able to format a string version of the `AuditRecord` object

with name value pairs of `type = MyRecord`, `severity = Failure`, and `message = Connection attempt failed`, resulting in a string version of `MyRecord/Failure/Connection attempt failed`.

Consider the following example:

```
AuditRecord aRec = new AuditRecord( "AUDITBASE", 5 /* FAILURE */,
"Connection attempt failed");

System.out.println( aRec.toString() );

// Outputs - "AUDITBASE/Failure/Connection attempt failed" as the string
version of aRec.
```

SAMPLEAUDITRECORD

The `SAMPLEAUDITRECORD` naming authority uses the colon character (`:`) character as the delimiter. This authority is expected to handle a name with `auditbase`, `arg1`, `arg2`, and `arg3` tokens.

An example name parsed by this naming authority is `MyRecordType/Info/Login successful:one:two:three`, which will be parsed into name value pairs as follows:

- `type = MyRecordType`
- `severity = Info`
- `message = Login successful`
- `arg1 = one`
- `arg2 = two`
- `arg3 = three`

Naming Authority Classes

You use the following classes to implement naming authorities:

- "AttributeValueEnumeration" on page 3-5
- "NameAttributeType" on page 3-6
- "NameAttributeValue" on page 3-6
- "NamingAuthority" on page 3-7

- “NamingAuthorityManager” on page 3-7
- “NamedObjects” on page 3-6
- “PolicyDomain” on page 3-7

In particular, the `com.bea.security.NamingAuthorityManager` class is a central registrar for all naming authorities that the Java Security Service Module recognizes. Within a Java Security Service Module, if the name is not registered with its naming authority's manager, then the name effectively does not exist.

This class is also responsible for managing the dependencies of the naming authorities. If a naming authority's dependencies are not met, then it will not be available to the Java Security Service Module runtime. The naming authority remains associated with the Naming Authority Manager, and every time a new authority is added, its dependencies are evaluated and enabled after the dependencies are all met.

This class is also responsible for loading XML definitions of authorities and validating them.

Attribute Precedence

The `NamingAuthority` base class provides parsing services for all naming authorities that define a name in a valid way using attributes. Attribute precedence comes into play when a name contains fewer delimited pieces then there are attributes defined within the naming authority. During token enumeration, certain attributes must be discarded to properly map the remaining attributes to the relevant values from the name.

Note: Application developers may choose to implement their own naming authority to override this default behavior from the base class.

In the `NamingAuthority` base class, there is a specific order in which attributes are selected for elimination. Attributes are trimmed either right-to-left or left-to-right. For specific information on each type of attribute, see Table 4-1.

Table 4-1 The Order in which Attributes are Selected for Elimination

Attribute	Trim Direction	Placement and Order of Selection for Elimination
`MULTI_TOKEN`	Right-to-left	Multi-token attributes consist of leftover tokens after all the other attributes are evaluated. Consequently, this attribute is the first one trimmed from a parsed name.

Table 4-1 The Order in which Attributes are Selected for Elimination (Continued)

SINGLE_VALUE	See next column	Single-value attributes assume their identity by place value. They can be placed either next to a PREFIX attribute, a TERMINAL attribute, or at the start of a name. • Single value attributes that are placed in relation to a PREFIX attribute are trimmed from left to right. • Single value attributes that are placed in relation to a TERMINAL attribute are trimmed from right to left • Single value attributes that are in name definitions that do not contain PREFIX or TERMINAL attributes are trimmed left to right.
MULTI_VALUE and SINGLE_VALUE TERMINAL	Right-to-left	Both multi value and single value terminal attributes are treated with equal precedence in a name.
MULTI_VALUE and SINGLE_VALUE PREFIX	Left-to-right	Both multi-value and single-value prefix attributes are treated with equal precedence in a name.

Example Naming Authority Definition

An example naming authority definition is shown in Listing 4-1.

Listing 4-1 Example Naming Authority Definition

```
<AuthorityDefinition name="ARME_RESOURCE_AUTHORITY" delimiters="/">
  <Attribute name="app" type="SINGLE_VALUE_PREFIX"/>
  <Attribute name="object" type="MULTI_TOKEN"
authority="ARME_RESOURCE_AUTHORITY"/>
</AuthorityDefinition>
```

Listing 4-1 describes a built-in naming authority called ARME_RESOURCE_AUTHORITY. This authority has two attributes. The first attribute is called app and it is of type

SINGLE_VALUE_PREFIX. The second attribute is called object and it is of type MULTI_TOKEN. The object token specifies authority ARME_RESOURCE_AUTHORITY itself for parsing it.

CHAPTER 5

Java Security Service Module APIs

To develop Java Security Service Module (SSM) applications, you use the Java interfaces and classes developed by BEA Systems and by Sun Microsystems, Inc.

This section covers the following topics:

Java Security Service Module APIs

The `com.bea.security` package contains all the BEA developed interfaces, classes, and exceptions that are supported by the Java Security Service Module.

This section lists and describes the key interfaces and classes that you use to develop Java Security Service Module applications. For a complete listing and description of the interfaces, classes, and exceptions included in the `com.bea.security` package, see *Javadocs for Java API*.

The key interfaces and classes are described in the following topics:

AuthenticationService API

The authentication service provides functions to an application related to establishing, verifying, and transferring a person or process identity. Thus, the authentication service provides two main functions: authentication and identity assertion.

The type of authentication supported is determined by the providers configured for the Java Security Service Module through the Administration Application. Each provider requires the Java application to respond to a callback specific to the type of authentication used by the Java application. The application is responsible for collecting the necessary credentials through user interaction.

The type of assertion token supported is also determined by what providers are configured in the Java Security Service Module runtime through the Administration Application. Each provider may implement one or more identity assertion token types.

An additional feature related to identity assertion is the ability for an identity assertion provider to provide a challenge first, which the Java Security Service Module application then has to process in an implementation dependant way before redeeming that token for an identity. Not all assertion providers support this challenge capability so knowledge of the configured provider is required when using this feature.

The `AuthenticationService` API is intended to provide a minimum level of limitation on the Java Security Service Module application concerning the use of features embodied by the authentication providers. Certain elements of this API are implemented according to the standard established by the Java Authentication and Authorization Service (JAAS). The use of the standard JAAS `CallbackHandler` interface and of the standard JAAS callbacks themselves provides a common and familiar means for application developers to authenticate users.

Elements of the identity assertion functions in this API rely on an assertion token name and an object reference. It is the responsibility of the authentication provider to choose suitably unique token names to represent the various types of tokens that it supports. It is possible, however, to have several authentication providers that support tokens described by the same token name. However, if these providers are configured within the same Security Service Module configuration for authentication, the results are undetermined, so it is not recommended that this be done. The object reference associated with an identity assertion token name is merely a way to pass a relevant object to the appropriate provider.

Note: A built-in identity asserter is included as part of the java api. This mechanism provides an easy way to convert between identity formats.

This asserter takes a token name of `WLS.Subject` and a token object that is the WebLogic Server form of identity (`javax.security.auth.Subject`) and authenticates and validates the subject. If an identity assertion provider is plugged in to handle the assertion type of `WLS.Subject`, then the provider is used. If no identity assertion provider is plugged in to handle this type, the built-in identity asserter is used.

For more information on the `AuthenticationService` API and the methods it supports, see *Javadocs for Java API*.

AuthorizationService API

The authorization service is a service that allows an application to determine if a specific identity is permitted to access a specific resource. This decision may then enforced in the application directly at the policy enforcement point.

This service revolves primarily around variations of the `isAccessAllowed()` method. At a minimum, this method collects information about the identity of the accessor, or requestor, the resource the accessor is attempting to accessed, and the action the accessor is attempting to perform on that resource. The longer form of this method supports passing in an `AppContext` object that makes available name/value pairs that security providers may use in their security evaluation.

The `isAccessAllowed()` method always requires that a valid, authenticated identity be present when requesting an access decision. For more information about using this method, see the *Javadocs for Java API* for the `com.bea.AuthorizationService.isAccessAllowed()` method.

The `AuthorizationService` API also provides the `isAuthenticationRequired()` method. This method provides a mechanism that an application can use to test to see whether the Java Security Service Module requires authentication to access any specific resource.

AuditingService API

The auditing service allows an application to log events based on activity related to enterprise security. The Java Security Service Module runtime uses these mechanisms to log appropriate data when events occur.

The `AuditingService` API is based on an event model. When something of note occurs, an application can derive a new class from the `AuditRecord` class or use the `AuditRecord` directly as it is a named object.

The Auditing Service is based on a single method, `recordEvent()`. This method accepts an object of the type `AuditRecord`, which encapsulates the logged information, and then passes it to the Java Security Service Module runtime. Based on its configuration, the Security Service Module runtime routes the event to the proper auditing providers so that it can be recorded.

For more information on the AuditingService API and the methods it supports, see *Javadocs for Java API*.

RoleService API

The role service allows an application to extract role information about specific identities and resources within the context of the application. These roles can then be used for customizing an interface or other purposes.

Note: Do not use roles themselves as authorization, because many policies allowing or disallowing access to a resource may be written against a role. Use the Authorization Service to determine actual rights.

The Role Service evaluates an interaction of an identity with a resource within an application context and returns a list of role names associated with the configuration of that identity. These roles can change with every resource or be static for the identity across all resources. The roles assigned to an identity are determined by the policy written within the policy domain.

The RoleService requires that the application pass in a valid identity, a valid resource, and a valid action. The application context is optional and may be set to `null` if no context is passed in.

The `RoleService` API has one key method that supports its primary function, `getRoles()`. This method gets the roles for an `AuthenticIdentity` in reference to a `RuntimeResource`, `RuntimeAction`, and an optional `AppContext`. Table 5-1 describes the `getRoles()` method parameters.

Table 5-1 getRoles Method Parameters

Parameter	Description
`AuthenticIdentity ident`	Identifies the user with which the roles are associated.
`RuntimeResource` resource	Represents the Java Security Service Module runtime resource for which the roles are present.

Table 5-1 getRoles Method Parameters

Parameter	Description
`RuntimeAction action`	Represents the Java Security Service Module runtime action for which the roles are relevant.
`AppContext context`	Specifies an object with an `AppContext` interface that contains the name/value pairs that represent this application's current context.

The `getRoles()` method returns a vector of `IdentityRole` objects that lists the roles associated for this identity. If the identity provided is invalid or not properly authenticated, this method throws an exception.

For more information on the `RoleService` API and the methods it supports, see *Javadocs for Java API*.

CredentialMappingService API

The credential mapping service allows an application to fetch credentials of certain types that are associated with a specific identity for a specific resource. These credentials can then be used on behalf of that identity to provide some privileged function, such as logging into a database or sending e-mail.

The `CredentialMappingService` API has one key method that supports its primary function, `getCredentials()`. This method gets a set of credentials relevant to a specific resource and action for an AuthenticIdentity. Table 5-2 describes the `getCredentials()` method parameters.

Table 5-2 getCredentials Method Parameters

Parameter	Description
`AuthenticIdentity byident`	Identifies the AuthenticIdentity of the user requesting this credential set.
`AuthenticIdentity forident`	Identifies the AuthenticIdentity of the user for whom the credentials are being fetched

Table 5-2 getCredentials Method Parameters

Parameter	Description
`RuntimeResource resource`	Represents the Java Security Service Module runtime resource relevant to this credential mapping.
`RuntimeAction action`	Represents the Java Security Service Module runtime action relevant to this credential mapping.
`java.lang.String[] credtypes`	Specifies an array of `Strings` containing the credential types requested.

The `getCredentials()` method returns a vector of credentials relevant for the types requested as follows:

- Password-based credentials are returned as an instance of the `javax.resource.spi.security.PasswordCredential` class.
- Other credentials are returned as a class that implements the `javax.resource.spi.security.GenericCredential` interface.

Note: Additional credential formats can also be returned. The caller is responsible for detecting and casting the credential formats to the appropriate Java class.

The user identities passed in using the `getCredentials()` method can contain any names, valid or invalid. If credentials are present matching that name (case sensitive), then they are returned. Any names requested, but not available, are ignored. This presents the possibility of an empty vector being returned with no exception thrown.

The `getCredentials()` method accepts an identity for the user for whom credentials are requested and an identity for the user querying for the credentials. Thus, the `getCredentials()` method must have two authenticated identities. If either of the identities is not valid or not properly authenticated, an exception is thrown. Additionally, the security policy must permit the user requesting the credentials to fetch the credentials of the other identity, or to fetch the credentials for themselves if they one and the same identity.

For more information on the `CredentialMappingService` API and the methods it supports, see *Javadocs for Java API*.

Java SDK APIs

The following Java SDK APIs can be used with the BEA AquaLogic Enterprise Security API:

- Sun Java 2 SDK 1.4.2_08 on WebLogic Server 8.1
- Sun Java 2 JDK 5.0 (JDK 1.5) on WebLogic Server 9.1 or 9.2
- BEA JRockit 1.4.2_08 SDK on WebLogic Server 8.1, on Windows or Linux
- BEA JRockit 5.0 (JDK 1.5) on WebLogic Server 9.1 or 9.2, on Windows or Linux

CHAPTER 6

Developing Applications Using the Java Security Service Module

The following topics are discussed in this section:

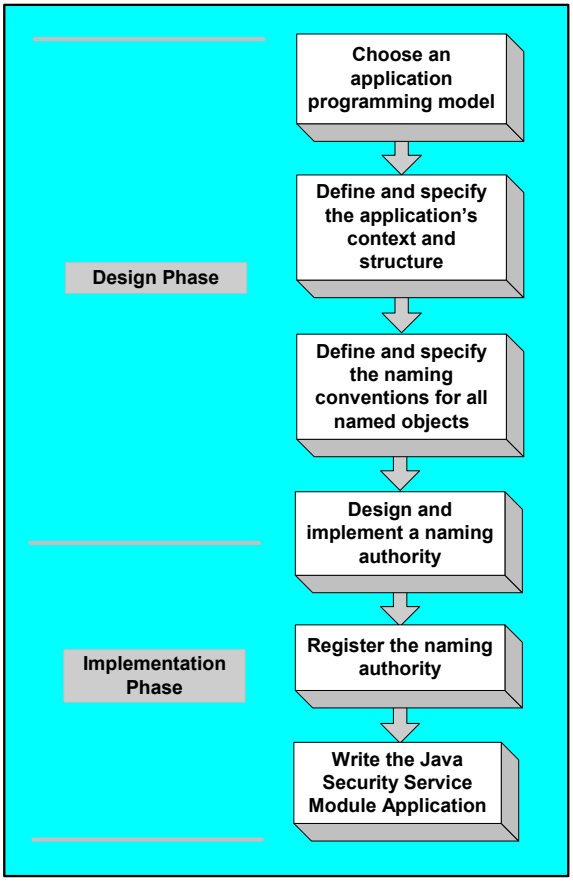
Overview of the Application Programming Steps

To use the Java Security Service Module to design and develop of an Java application, perform the following steps (see Figure 6-1):

1. Choose an application programming model. For instructions on how to perform this task, see "Choosing an Application Programming Model" on page 6-3.
2. Define and specify the application context and structure. For instructions on how to perform this task, see "Defining the Application Context and Structure" on page 6-4.
3. Define and specify the naming conventions for all named objects. For instructions on how to perform this task, see "Defining and Specifying Naming Conventions" on page 6-5.

4. Design and implement a naming authority. For instructions on how to perform this task, see "Defining and Implementing a Naming Authority" on page 6-5.
5. Register the naming authority. For instructions on how to perform this task, see "Registering a New Naming Authority" on page 6-7.
6. Write the Java Security Service Module Application. For instructions on how to perform this task, see "Writing Java Security Service Module Applications" on page 6-9.

Figure 6-1 Java Security Service Module Application Development Process



Choosing an Application Programming Model

When you develop your application, you have a choice of two different programming models: an open model or a closed model.

If an application conforms to the closed model, when a user attempts to use your application, the application responds by querying the user to find out who they are and then authenticates them. From that point forward, the application always knows the user identity. Thus, the application is closed in the sense that once the user is authenticated, their identity is established and rigid rules can then be enforced. The application uses that user identity for that user until the user logs out of the application. If an application conforms to the open model, the application only asks who the user is when there is a need to know.

The difference between the two models is that in the closed model, the application immediately does user authentication and never has to call the `com.bea.security.AuthorizationService.isAuthenticationRequired()` method on behalf of the user to determine whether a requested resource is protected. The application simply uses the identity of the user to call the `com.bea.security.AuthorizationService.isAccessAllowed()` method to determine if the authenticated user has the privileges required to access to resource.

In the open model, however, every time the unauthenticated user requests access to a resource, the application must call the `isAuthenticationRequired()` method. If the method returns `False` indicating that the resource does not require the application to authenticate users, the application allows the user to access it—the user is a guest and the resource is open to the public. If the `isAuthenticationRequired()` method returns `True`, then the application must authenticate the user to find out who they are and then call the `com.bea.security.AuthorizationService.isAccessAllowed()` method on behalf of the user to determine if the authenticated user has the privileges required to access the requested resource.

Both programming models are valid. The model you choose depends upon the nature of your application. The following scenarios describes typical of each type of programming model, open and closed.

- ***Open Programming Model Scenario***—A newspaper web site is an example of an application in which you would use the open programming model. When users access the site, they are allowed to look at current news stories from the most recent issue of the newspaper without being asked to authenticate, or login. This is because access to current news stories is available to the public. However, if users request access to the newspaper archives, information to which access is restricted to subscribers only, they are prompted to

supply credentials (such as a username and subscriber ID combination) so that they can be authenticated. Once authenticated, they are allowed to view the archives.

- ***Close Programming Model Scenario***—A banking web site is an example of an application in which you would use the close programming model. When users access the site, they are immediately required to authenticate, or login. This way, no one can access any information on the site without authenticating first. Once users are authenticated, they are allowed to access, however, their access is restricted to viewing their accounts and performing transactions on their accounts. Only users who have special privileges, such as bank tellers, are allowed access to accounts other then their own.

Defining the Application Context and Structure

An context of an application is the environment within which the application functions. The context allows you to write rules that use elements in this context. The context includes a collection of names and values that you implement using the `AppContext` interface. Once the interfaces are implemented, you pass the resulting objects through to the security providers. The security providers can then query the interfaces as they interact with the application.

To define the applications context and structure, do the following:

1. Define the application context

   The application context establishes the user environment. For example, for a user session, the context defines the information about the user that the application tracks such as time spent using the application and details about the user.

   Business and security manages usually collaborate to write rules that are then used to define the collection of names and values implemented in the `AppContext` interface. For example, their could be a rule for granting users different levels of authorization based on their financial status and specifying the requirements for being granted each authorization level. Another rule could assign users certain capabilities based on their credentials as determined by a credential mapper. If the application detects that a user has asserted their identity from a SAML token, then the application can make a SAML artifact available to pass their identity on and obviate the need to re-authenticate themselves.

2. Use the `AppContext` interface to implement the collection of names and values.

3. Structure your resources.

   It is beneficial to create a hierarchy of resources so that resources have a parent to child relationship. This allows you to write rules to take advantage of inheritance. One example of this is having layers of forms in an Administration Application. In this case, you may

have a series of items such as `Console.FORMNAME.menu.dropdown.menuitem`, `Console.FORMNAME.body.fieldname`, `Console.FORMNAME.body.header.fieldname`, and so on. By segmenting your application into these related hierarchical names, rules can be applied to the parent (`Console.FORMNAME`) or a child of the parent (`Console.FORMNAME.body`), allowing for a rule exception model.

Defining and Specifying Naming Conventions

There are four types of named objects used in Java Security Service Module applications: `actions`, `auditRecords`, `ContextAuditRecord`, and `resources`. Because named objects conform to naming conventions, the security providers can look up their structure in the naming authority.

You must make decisions about the naming conventions to use for these objects and the naming authorities that are used to define and control the structure of their names. Some specific questions that you need to answer about named objects are as follows:

- How am I going to name my resources? Which naming scheme do I use?

  URL is an example of a naming scheme.

- What does a name consist of?
- Will names use colons or slashes as delimiters?
- Will I use sub-names so as to allow a resource name to contain other names?
- How will I implement the naming authority? Will the application use the pre-configured URLBASE naming library that is provided with the Java Security Service Module?
- How will I load the naming authority XML format?

Defining and Implementing a Naming Authority

To design and implement a naming authority, perform the following steps:

1. Pick a name for the naming authority and make a decisions on the naming authority delimiter, attributes, sub-authorities, and dependencies.
2. Decide which mechanism to use to define the naming authority. You have a choice of three mechanisms, as follows:
   - Use the `DataDrivenAuthority` base class. For instructions on using this mechanism, see "Using the DataDrivenAuthority Class" on page 6-6.

- Use an XML file. For instructions on using this mechanism, see "Using a DataDrivenAuthority XML File" on page 6-6.
- Create a custom naming authority class. For instructions on using this mechanism, see "Creating a Custom Naming Authority Class from the NamingAuthority Base Class" on page 6-7.

3. Register the naming authority with the Naming Authority Manager. For instructions on registering the naming authority, see "Registering a New Naming Authority" on page 6-7.

Using the DataDrivenAuthority Class

The `com.bea.security.DataDrivenAuthority` class provides a programmatic way to define a naming authority. By giving the `DataDrivenAuthority` class a name and adding attributes, delimiters, dependencies, and sub-authorities, you can define new naming conventions and construct a fully functional naming authority.

Using a DataDrivenAuthority XML File

You can use an XML file to define a name authority within an XML document so that the Naming Authority Manager constructs your naming authority for you. The XML file shown in Listing 6-1 illustrates an example of a XML file that can be loaded into the Naming Authority Manager to automatically build instances of a `DataDrivenAuthority` class to govern naming systems.

Listing 6-1 DataDrivenAuthority XML File

```
<?xml version="1.0"?>
<AuthorityConfig>
   <!-- *** Comment *** -->
   <AuthorityDefinition name="NAME" delimiters=":">
      <Dependencies>
          <Authority name="OTHERAUTH"/>
          <Authority name="YETANOTHERAUTH"/>
      </Dependencies>
      <Attribute name="singlename" type="SINGLE_VALUE_PREFIX"/>
      <Attribute name="multiname" type="MULTI_VALUE_TERMINAL"
                 authority="NAME:SUB"/>
      <!-- *** Another Comment *** -->
      <AuthorityDefinition name="NAME:SUB" delimiters="&amp;">
```

```
            <Attribute name="singleterm" type="SINGLE_VALUE_TERMINAL"/>
            <Attribute name="multiterm" type="MULTI_VALUE_TERMINAL"/>
         </AuthorityDefinition>
      </AuthorityDefinition>
      <!-- *** Yet Another Comment *** -->
      <AuthorityDefinition class="com.my.packagename.MyAuthority"/>
</AuthorityConfig>
```

In this example, the embedded `AuthorityDefinition` tags define sub-authorities. However, sub-authority names are still manually named to provide flexibility. Also, there is a direct reference to a class in the `CLASSPATH` with a simple `class="classname"` in an `AuthorityDefinition` tag. This class must base the `com.bea.security.NamingAuthority` class.

Creating a Custom Naming Authority Class from the NamingAuthority Base Class

You may create a custom class that is derived from the `com.bea.security.NamingAuthority` base class. You can use this custom class to override the methods of the base class and provide custom or optimized parsing.

Only consider creating a custom naming authority class if one of the following scenarios applies:

- If the naming structure is so complicated that it is difficult to represent using the standard attributes and functions of the base class.
- If the naming structure is so complicated that there are time issues with using the standard parsing methods.

Note: If you decide to create a custom naming authority class, there are several functions that you can choose to implement in Java code rather than use the recursive authority/sub-authority mechanism used by the naming authority base class.

Registering a New Naming Authority

Before your Java application can use a new naming authority, you must register the naming authority with the Naming Authority Manager.

When a naming authority is registered, all sub-authorities that it references are also registered. In addition, all naming authorities and sub-authorities are validated to ensure that they have proper attribute definitions and that their dependencies are satisfied. If each naming authority has a proper attribute configuration and its dependencies are met, it is tagged as validated and can be retrieved from the Naming Authority Manager for use.

If a naming authority does not pass validation, it is marked as invalid. Java applications cannot retrieve invalid naming authorities from the Naming Authority Manager. Every time a new naming authority is registered with the Naming Authority Manager, all invalid naming authorities are re-evaluated for dependencies and marked as valid if their dependencies are met.

The procedure you use to register a naming authority with the Naming Authority Manager differs depending on whether you are using a naming authority class or an XML file to define your naming authority. The following topics describe the different procedures:

- "Using a Naming Authority Class" on page 6-8.
- "Using an XML file" on page 6-8.
- "Using a DataDrivenAuthority Object" on page 6-9.

Using a Naming Authority Class

To register your naming authority class programmatically, fetch an instance of the Naming Authority Manager from the Java Security Service Module and pass your naming authority in using the `com.bea.security.NamingAuthorityManager.registerAuthority` method. If the Naming Authority Manager contains an existing naming authority with the same name as the one you are registering, the existing naming authority is replaced by the newly registered naming authority, provided the new naming authority validates.

Using an XML file

If you defined your naming authority in an XML file, use either of the following methods to register it.

- **Method 1:** Register the XML file with your application configuration object, `AppConfig`, so that when the `AppConfig` object is passed in to your application during Java Security Service Module initialization, the XML file is loaded.
- **Method 2:** Call the `com.bea.security.NamingAuthorityManager.loadXMLAuthorityDefinition()` method and pass in the name of the XML file to the Naming Authority Manager.

- The Naming Authority Manager reads the file and creates an instance of the `DataDrivenAuthority` object and registers it.

Using a DataDrivenAuthority Object

To register a naming authority using a `DataDrivenAuthority` object, programmatically construct a `DataDrivenAuthority` object, passing in a name, delimiters, attributes, sub-authorities, and dependencies, and then register that object with the Naming Authority Manager using the `com.bea.security.NamingAuthorityManager.registerAuthority()` method.

Writing Java Security Service Module Applications

This section covers the following topics:

Knowledge Required of the Java Security Service Module Environment

For the vast majority of security functions, you do not have to know about the configuration and capabilities of security providers. However, there are some functions that require that you know what the providers pass in and there is no way to query the security providers and have them tell you about themselves.The security providers pass objects as Java objects, so you must decide how to cast the object. For example, the object that you use to assert identity is determined by the identity asserter provider configured for your environment. If you have a SAML identity assertion provider configured, then it expects a SAML identity object to be passed in. The same is true with the Credential Mapper. A Credential Mapper defines two types of credentials that it passes, but the type received by the Credential Mapper really depends upon the type of Credential Mapping Provider configured. The credential is passed in as an object so you have to know if you asked for a credential. For example, an Oracle database password Credential Mapping Provider

passes in password credential objects. The same consideration applies to Authentication Providers. The Java Authentication and Authorization Service (JAAS) defines six callback types. However, you can create an Authentication Provider that provides a custom callback that deviates from the standard.

Note: The Java Security Service Module code examples provided in this document assume that the Security providers supplied with the product are configured in the Java Security Service Module environment.

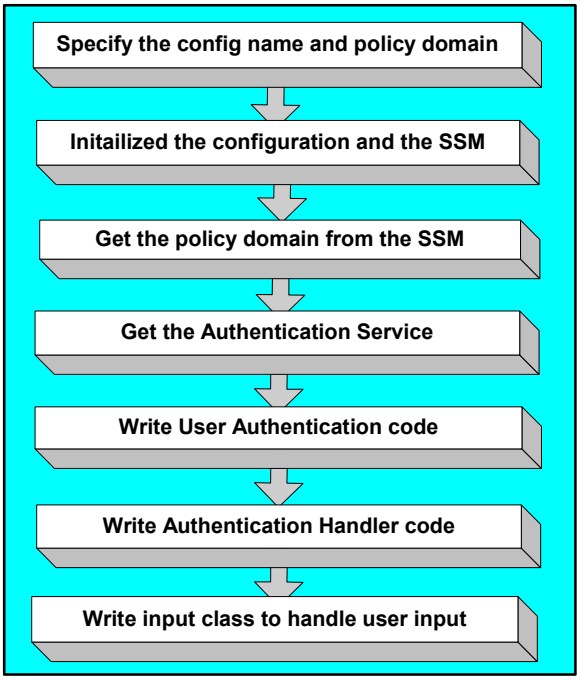
Writing an Authentication Application

This section covers the following topics:

Step-By-Step Procedure for Writing an Authentication Application

To write an authentication application, perform the following steps (see Figure 6-2):

Figure 6-2 Developing a Java Security Service Module Authentication Application



1. Write code to specify the configuration name and the policy domain. The configuration is created and named in the Administration Application. Note that a policy domain shares the same name as the configuration used to create it. (see Listing 6-2).

Listing 6-2 Policy Domain Specification Code

```
// The configuration name to use.
public static final String CONFIGNAME = "policydomain";

// The policy domain shares the same name as its configuration.
// The policy domain name to use.
public static final String POLICYDOMAIN = "policydomain";
```

2. Write initialization code to initialize the configuration of the application and to initialize the Java Security Service Module runtime to get an instance of it (see Listing 6-3).

Listing 6-3 Application Configuration and Java Security Service Module Initialization Code

```
// Initialize this applications configuration.
AppConfig cfg = new AppConfig();
// Policydomains are named after their configurations.
        cfg.useConfiguration( CONFIGNAME );
// Initialize the security runtime.
        try {
            SecurityRuntime.initialize( cfg );
        }
        catch( ParameterException pExc ) {
            // An error occurred with our configuration.
            System.out.println( pExc.getLocalizedMessage() );
            System.exit(pExc.hashCode());
        }
// Get an instance of the runtime
        SecurityRuntime rt = SecurityRuntime.getInstance();
        System.out.println("Security Runtime Initialized");
```

3. Write code to get the policy domain from the Java Security Service Module runtime (see Listing 6-4).

Listing 6-4 Get Policy Domain Code

```
        // Fetch the policy domain from the runtime.
        PolicyDomain pd = null;
        try {
             pd = rt.getPolicyDomain( POLICYDOMAIN );
        }
        catch( ParameterException pExc ) {
             // We could not get the policy domain.
             System.out.println( pExc.getLocalizedMessage() );
```

```
            System.exit(pExc.hashCode());
        }
        System.out.println("Retrieved Policy Domain");
```

4. Write code to get the Authentication Service from the policy domain (see Listing 6-5).

Listing 6-5 Get the Authentication Service Code

```
// Get the authentication service from the policy domain.
        AuthenticationService atnSvc = null;
        try {
            atnSvc = (AuthenticationService) pd.getService(
                                              ServiceType.AUTHENTICATION  );
        }
        catch( ServiceNotAvailableException naExc ) {
            // We could not fetch the service.
            System.out.println( naExc.getLocalizedMessage() );
            System.exit( naExc.hashCode() );
        }
        System.out.println("Retrieved Authentication Service");
```

5. Optional: Write code to check the service type to make sure that it is an Authentication Service (see Listing 6-6). If there are multiple versions of the Authentication Service available, you may also write code to check the version of the Authentication Service and verify that the version is compatible with the authentication application.

Listing 6-6 Authentication Service Type, Version, and isCompatible Code

```
// Validate that the service we retrieve is the type and level of
// compatibility we need.
// Get the policy domain.
PolicyDomain pd = null;
try {
    pd = rt.getPolicyDomain( "POLICYDOMAIN" );
```

```
}
catch( ParameterException pExc ) {
    System.err.println( pExc.getMessage() );
    System.exit(-10);
}

// Get the authentication service.
PublicSecurityService myService = null;

try {
    myService = pd.getService( ServiceType.AUTHENTICATION );
}
catch( ServiceNotAvailableException svcExc ) {
    System.err.println(svcExc.getMessage());
    System.exit(-10);
}

// Lets make sure that what is returned is an authentication
// service.

ServiceType sType = myService.getServiceType();

if ( sType != ServiceType.AUTHENTICATION ) {
   System.err.println("We did not receive an authentication service!");
   System.exit(-10);
}

// Lets make sure this service is compatible with the version we are coding
// against - version 1.0.

ServiceVersion expectedVersion = new ServiceVersion( 1, 0 );

int compatiblity_flag = myService.isCompatible( expectedVersion );

if ( compatibility_flag == PublicSecurityService.COMPATIBLE ) {
   // This service is fully compatible with this application
   // proceed with initialization.
} else if ( compatibility_flag ==
PublicSecurityService.COMPATIBLE_DEPRECATED ) {
   // This service is fully compatible with this application
   // except some functions will be changing in upcoming versions (some
   // functions have been deprecated between versions).
   // Warn the developers in a log file
```

```
   log_warning("Authentication Service may use deprecated methods in version
                                " + myService.getVersion() )
} else if ( compatibility_flag == PublicSecurityService
                                          .COMPATIBLE_UNKNOWN)
{
   // The compatibility of this service is unknown. Usually because we
   // expect a larger service version than what is currently installed.
   // Compatibility may be assumed (you can try it)
   // or you may choose to fail on unknown. We will warn the user
  popup_warning("Authentication Service expects version " + expectedVersion
+ " and got version " + myService.getVersion());
} else if ( compatibility_flass == PublicSecurityService.NOT_COMPATIBLE ) {
   // This service is known not to be compatible with the version expected.
   // Stop initialization and exit this application
   popup_warning("Authentication Service version " + myService.getVersion()
                           + " is not compatible with this application" );
   System.exit(-20);
}
// Continue with application initialization...
```

6. Write code to authenticate the user (see Listing 6-7).

Listing 6-7 User Authentication Code

```
// Start authentication
AuthenticIdentity ident=null;

try {
   ident = atnSvc.authenticate( new AuthenticationHandler() );
}
catch( IdentityNotAuthenticException naExc ) {
   System.out.println( naExc.getLocalizedMessage() );
   System.exit( naExc.hashCode() );
}
```

```
System.out.println("Authentication Succeeded");
System.out.println( ident.toString() );
```

7. Write code that creates an `AuthenticationHandler` class that implements `javax.security.auth.callback.CallbackHandler` interface (see Listing 6-8). This class loops through the callbacks, prompts the user for the appropriate input to gather credentials, and fills in the username and password.

Listing 6-8 Authentication Handler Code

```
public static class AuthenticationHandler implements
     javax.security.auth.callback.CallbackHandler {

   protected String username = null;
   protected char[] password = null;

   public String getPassword() {
        return new String( password );
   }

   public String getUsername() {
        return username;
   }

   public AuthenticationHandler( ) {
   }

   public AuthenticationHandler( String username, String password ) {
       this.username = username;
       this.password = password.toCharArray();
   }

   public void handle(Callback[] callbacks) throws IOException,
                    UnsupportedCallbackException {

      // Loop through the callbacks and prompt the user with the
                appropriate input to gather credentials

        for ( int numcb=0; numcb < callbacks.length; numcb++ ) {
```

```
            // Fill in the username
          if ( callbacks[numcb] instanceof
                   javax.security.auth.callback.NameCallback ) {
           javax.security.auth.callback.NameCallback ncb = (
         javax.security.auth.callback.NameCallback ) callbacks[numcb];
           if ( username == null ) {
           // Fetch the username from the user
            username = Input.getString( ncb.getPrompt() );
            ncb.setName( username );
             } else {
              ncb.setName( username );
              }
           }
           // Fill in the password
           if ( callbacks[numcb] instanceof
            javax.security.auth.callback.PasswordCallback ) {
               javax.security.auth.callback.PasswordCallback pcb = (
                    javax.security.auth.callback.PasswordCallback )
                              callbacks[numcb];
             if ( password == null ) {
             String pwd = Input.getString( pcb.getPrompt() );
             password = pwd.toCharArray();
             }
              pcb.setPassword( password );
           }
      }
   }
}
```

8. Write code that creates an input class to handle user input (see Listing 6-9).

Listing 6-9 User Input Class Code

```
public static class Input {
   public static String getString( String prompt ) {
      BufferedReader stdin = new BufferedReader( new InputStreamReader(
                                                    System.in ) );
       System.out.print( prompt );
       System.out.flush();
       String input = null;
       try {
         input = stdin.readLine();
       }
       catch( IOException ioExc ) {
            System.out.println( ioExc.getLocalizedMessage() );
       }
        if ( input == null ) {
            return "";
        } else {
            return input;
        }
     }
}
```

Other AuthenticationService Methods

The following `com.bea.security.AuthenticationService` class methods are also available for use in developing authentication applications:

- "assertIdentity Method" on page 6-19
- "getChallengeAsssertionToken Method" on page 6-20
- "isAssertionTokenSupported Method" on page 6-21

assertIdentity Method

The `assertIdentity()` method is used to establish an identity for a user through identify assertion. The token-type name passed into the Security Service Module must be supported by a provider or an exception is thrown. The object reference must implement an interface known to the token provider and must be a valid token or else an exception is thrown. If these requirements are met, this method returns an authenticated identity.

The format of the token is up to the Identity Assertion Provider that handles that token type In Listing 6-10, the token is a certificate and the token type is X.509. Therefore, to assert identity so as to satisfy the requirements noted in the previous paragraph, an application developer must understand the format of the token he passes in.

Listing 6-10 assertIdentity Method Code

```
// Get an X509 certificate from somewhere in my application.
     AppCertificate certificate = myApp.getCertificate();
     // Assert identity from the token.
     AuthenticIdentity myUser = null;
     try {
      myUser = authSvc.assertIdentity( "X509", certificate.getBytes() );
     }
     catch( IdentityNotAuthenticException inaExc  ) {
         // This exception is thrown if I am not allowed to assert
         // an identity from this token.
         System.err.println( inaExc.getMessage());
     }
     catch( InvalidAssertionTokenException inatExc ) {
       // This exception is thrown if the token type X509 is not
      // understood by the providers or if the token format is invalid.
         System.err.println( inaExc.getMessage());
     }
```

getChallengeAsssertionToken Method

The `getChallengeAsssertionToken()` method returns an identity assertion token that contains a challenge that needs a response from the application before asserting the identity (see Listing 6-11). If the token type requested is not a challenge token type, an exception is thrown. This method returns an object reference with an interface type appropriate to the token type requested and needs to be cast to the appropriate type before it is used.

Listing 6-11 getChallengeAsssertionToken Method

```
// Get the SKey challenge object.
Object token = null;
try {
    token = authSvc.getChallengeAssertionToken("S/KEY");
}
catch( InvalidAssertionTokenException inatExc ) {
    // This exception is thrown if the providers do not know
    // about a challenge token format of "S/KEY".
    System.err.println( inatExc.getMessage() );
    System.exit(-10);
}

// Caste the challenge token to the correct format.
SKEYChallenge chal = (SKEYChallenge) token;

// Challenge the user and get the response.
String response = challengeUser( chal.getChallenge() );

// Fill in the challenge token
chal.setResponse( response );

// Assert identity with the challenge token.
AuthenticIdentity myUser = null;
try {
    myUser = authSvc.assertIdentity( "S/KEY", chal );
}
catch( IdentityNotAuthenticException inaExc  ) {
    // This exception is thrown if I am not allowed to assert
    // an identity from this token.
    System.err.println( inaExc.getMessage() );
```

```
}
catch( InvalidAssertionTokenException inatExc ) {
  // This exception is thrown if the token type S/KEY is not
  // understood by the providers or if the token format is invalid.
    System.err.println( inatExc.getMessage() );
}
```

isAssertionTokenSupported Method

This method identifies whether a specific token type, indicated by its unique name, is available for use within current policy domain (see Listing 6-12). This method returns a `true` or `false`.

Listing 6-12 IsAssertionTokenSupported Method Code

```
// Query if the "X509" assertion type is supported.
      boolean isSupp = authSvc.isAssertionTokenSupported("X509");
    if ( isSupp ) {
      System.out.println("X509 assertion tokens are supported");
    } else {
      System.out.println("X509 assertion tokens are NOT supported");
    }
```

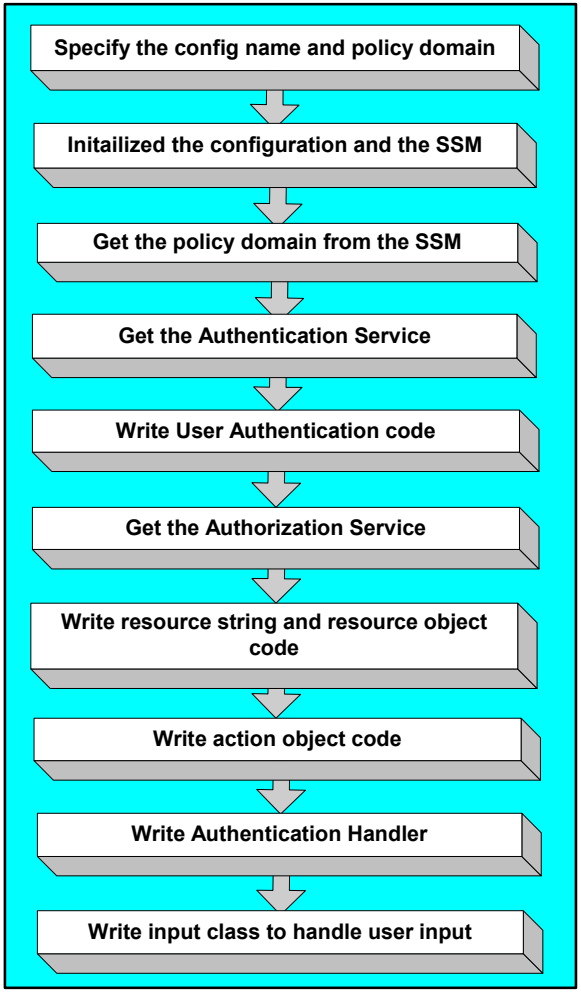
Writing an Authorization Application

This section covers the following topics:

Step-by-Step Procedure for Writing an Authorization Application

To write an authorization application, perform the following steps (see Figure 6-3):

Figure 6-3 Developing a Security Service Module Authorization Application



1. Write code to specify the configuration name and the policy domain. The configuration is created and named in the Administration Application. Note that a policy domain shares the same name as the configuration used to create it. For a code fragment that shows how to program this step, see Listing 6-2, “Policy Domain Specification Code,” on page 6-11.

2. Write initialization code to initialize the configuration of the application and to initialize the Security Service Module to get an instance of it. For a code fragment that shows how to program this step, see Listing 6-3, “Application Configuration and Java Security Service Module Initialization Code,” on page 6-12.
3. Write code to get the policy domain from the Java Security Service Module. For a code fragment that shows how to program this step, see Listing 6-4, “Get Policy Domain Code,” on page 6-12.
4. Write code to get the Authentication Service from the policy domain. For a code fragment that shows how to program this step, see Listing 6-5, “Get the Authentication Service Code,” on page 6-13.
5. Optional: Write code to check the service type to make sure that it is an Authentication Service. If there are multiple versions of the Authentication Service available, you may also write code to check the version of the Authentication Service and to verify that the version is compatible with the authorization application. For a code fragment that shows how to program this step, see Listing 6-6, “Authentication Service Type, Version, and isCompatible Code,” on page 6-13.
6. Write code to authenticate the user. For a code fragment that shows how to program this step, see Listing 6-7, “User Authentication Code,” on page 6-15.
7. Write code to get the Authorization Service from the policy domain (see Listing 6-13).

Listing 6-13 Get Authorization Service Code

```
// Authentication complete - now authorize a resource.
// Get the authorization service from the policy domain.

AuthorizationService atzSvc = null;

try {

    atzSvc = (AuthorizationService) pd.getService(
                          ServiceType.AUTHORIZATION  );
}
catch( ServiceNotAvailableException naExc ) {
    // We could not fetch the service
    System.out.println( naExc.getLocalizedMessage() );
    System.exit( naExc.hashCode() );
```

```
        }
        System.out.println("Retrieved Authorization Service");
```

8. Optional: Write code to check the service type to make sure that it is an Authorization Service. If there are multiple versions of the Authorization Service available, you may also write code to check the version of the Authorization Service and verify that the version is compatible with the authorization application. To program this step, use the code fragment shown in Listing 6-6, "Authentication Service Type, Version, and isCompatible Code," on page 6-13 and substitute Authorization for Authentication wherever references to the AuthenticationService occur.
9. Write code to request a resource string in the format defined by naming authority, create the resource object, output the resource for user confirmation, and request an action string in the format defined by action authority (see Listing 6-14).

Listing 6-14 Request Resource String and Create Resource Object Code

```
        // Output the resource string defined by the naming authority.
        System.out.println("EXAMPLERESAUTHORITY is defined as:
            servername:user@filename-fieldname (i.e.
                        www:bob@index.html-searchinput )");
         String resourcestring = Input.getString("Enter a resource string in
                        the format defined by EXAMPLERESAUTHORITY: ");
        // Create the resource object
        RuntimeResource rResource = new RuntimeResource( resourcestring,
                                            "EXAMPLERESAUTHORITY" );
System.out.println("EXAMPLEACTAUTHORITY is defined as:
                               mainaction:followup (i.e. GET:LOG)");
        String actionstring = Input.getString("Enter an action string
                             in the format defined by EXAMPLEACTAUTHORITY: ");
```

10. Write code to create the `RuntimeAction` object and output the parsed resource for user confirmation. In Listing 6-15, note that the `isAccessAllowed()` method is used to

determine whether the authenticated user (`identi`) is authorized to perform the requested `action` on the designated `resource`.

Listing 6-15 Action Object Code

```
        // Create the action object.
        RuntimeAction rAction = new RuntimeAction( actionstring,
                                                "EXAMPLEACTAUTHORITY" );

        // Output the parsed action for user confirmation.

System.out.println("Authorizing user for this resource and action\n");

        AccessResult rResult = null;
        try {
             rResult = atzSvc.isAccessAllowed( ident , rResource, rAction );
        }
        catch( IdentityNotAuthenticException idExc ) {
            System.out.println( idExc.getLocalizedMessage() );
            System.exit( idExc.hashCode() );
         }

         // We got a valid result
        System.out.println("Access Result:");
        System.out.println("   Access Allowed: " + String.valueOf(
                                rResult.isAllowed() ) );
        System.out.println("   Decision Time: " +
                                 rResult.getDecisionTime().toString() );
}
```

11. Create an `AuthenticationHandler` class that implements `javax.security.auth.callback.CallbackHandler` interface. This class loops through the callbacks, prompts the user for the appropriate input to gather credentials, and fills in the username and password. For a code example that shows how to program this step, Listing 6-8, “Authentication Handler Code,” on page 6-16.

12. Create an input class to handle user input. For a code fragment that shows how to program this step, see Listing 6-9, “User Input Class Code,” on page 6-18.

AuthorizationService.isAuthenticationRequired Method

The `com.bea.security.AuthorizationService.isAuthenticationRequired()` method is used in developing authorization applications. Listing 6-16 shows how to program `isAuthenticationRequired()` method. This method returns `true` or `false` to indicate whether an anonymous user is required to authenticate before accessing a resource and action. If this method returns `false`, user authentication is not required and no further access query needs to be performed. If this method returns `true`, then the application must authenticate the user and then call the `isAccessAllowed()` method on the AuthorizationService to get an access decision. For a code fragment that shows how to use the `isAccessAllowed()` method, see Listing 6-15, "Action Object Code," on page 6-25.

Listing 6-16 isAuthenticationRequired Method

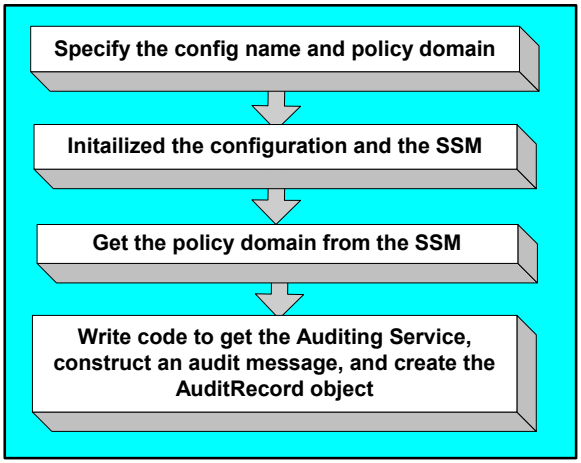
```
// Determine if authentication is required to access the URL
// http://www.bea.com/index.html via an HTTP GET.

// Create the resource.
RuntimeResource beaurl = new RuntimeResource(
                      "http://www.bea.com/index.html" , "URL_AUTHORITY" );

// Create the action.
RuntimeAction action = new RuntimeAction( "GET", "SIMPLE_ACTION" );

// See if authentication is required.
boolean result = atzService.isAuthenticationRequired( beaurl, action );

if ( result == true ) {
   // Authentication is required. I need to authenticate the user
   // and then call isAccessAllowed on this authorization service
   // before I can serve this URL to the user
   ...
} else {
   // Authentication is not required. I can allow this user to access this
   // URL.
   ...
}
```

Writing an Auditing Application

This section provides a step-by-step procedure for writing an auditing application.

To write an auditing application, perform the following steps (see Figure 6-4):

Figure 6-4 Developing a Java Security Service Module Auditing Application



1. Write code to specify the configuration name and the policy domain. The configuration is created and named in the Administration Application. Note that a policy domain shares the same name as the configuration used to create it. For a code fragment that shows how to program this step, see Listing 6-2, “Policy Domain Specification Code,” on page 6-11.
2. Write initialization code to initialize the configuration of the application and to initialize the Java Security Service Module to get an instance of it. For a code fragment that shows how to program this step, see Listing 6-3, “Application Configuration and Java Security Service Module Initialization Code,” on page 6-12.
3. Write code to get the policy domain from the Java Security Service Module. For a code fragment that shows how to program this step, see Listing 6-4, “Get Policy Domain Code,” on page 6-12.
4. Write code that gets the Auditing Service from the policy domain, constructs an audit message, and creates the `AuditRecord` object to pass the auditing event to the security runtime (see Listing 6-17).

Listing 6-17 AuditingService Code

```
// Get the audting service.
 PublicSecurityService myService = null;

 try {
     myService = pd.getService( ServiceType.AUDIT );
 }
 catch( ServiceNotAvailableException svcExc ) {
   // Exception is thrown if service is not present
     System.err.println(svcExc.getMessage());
     System.exit(-10);
 }

 AuditingService adtService = (AuditingService) myService;

 // Create an audit record to determine it's naming authority,
 // severity, and message.

AuditRecord adtRec = new AuditRecord("MyAppError", AuditRecord.ERROR,
                                          "This is an ERROR message");

 // Record the audit record to the auditing service provider
 adtService.recordEvent( adtRec );
```

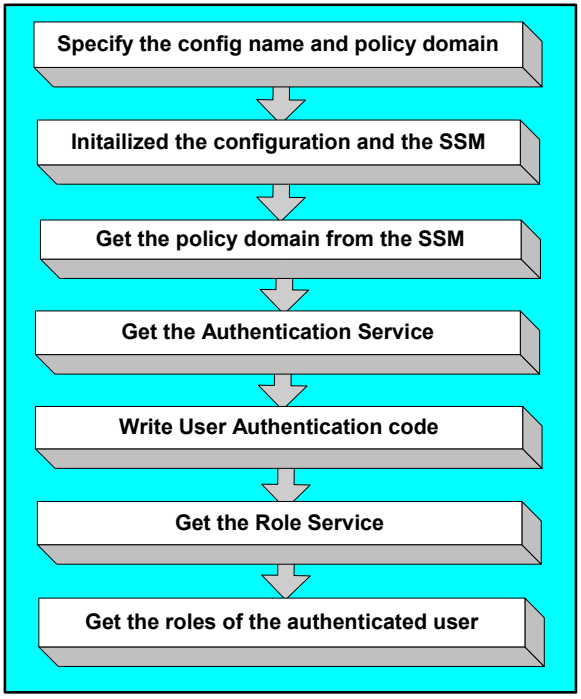
5. Optional: Write code to check the service type to make sure that it is an Auditing Service. If there are multiple versions of the Auditing Service available, you may also write code to check the version of the Auditing Service and to verify that the version is compatible with the authorization application. To program this step, use the code fragment shown in Listing 6-6, "Authentication Service Type, Version, and isCompatible Code," on page 6-13 and substitute Auditing for Authentication wherever references to the AuthenticationService occur. After writing this `type`, `version` and `iscompatible` code, insert it into code in Listing 6-17 just after the code that gets the Auditing Service.

Writing a Role Service Application

You use the role service to allow the application to extract role information about specific identities and resources within the context of the application.

To write a Role Service application, perform the following steps (see Figure 6-5):

Figure 6-5 Developing a Java Security Service Module Role Service Application



1. Write code to specify the configuration name and the policy domain. The configuration is created and named in the Administration Application. Note that a policy domain shares the same name as the configuration used to create it. For a code fragment that shows how to program this step, see Listing 6-2, “Policy Domain Specification Code,” on page 6-11.
2. Write initialization code to initialize the configuration of the application and to initialize the Java Security Service Module to get an instance of it. For a code fragment that shows how to program this step, see Listing 6-3, “Application Configuration and Java Security Service Module Initialization Code,” on page 6-12.
3. Write code to get the policy domain from the Java Security Service Module. For a code fragment that shows how to program this step, see Listing 6-4, “Get Policy Domain Code,” on page 6-12.

4. Write code to get the Authentication Service from the policy domain. For a code fragment that shows how to program this step, see Listing 6-5, “Get the Authentication Service Code,” on page 6-13.
5. Optional: Write code to check the service type to make sure that it is an Authentication Service. If there are multiple versions of the Authentication Service available, you may also write code to check the version of the Authentication Service and to verify that the version is compatible with the authorization application. For a code fragment that shows how to program this step, see Listing 6-6, “Authentication Service Type, Version, and isCompatible Code,” on page 6-13.
6. Write code to authenticate the user. For a code fragment that shows how to program this step, see Listing 6-7, “User Authentication Code,” on page 6-15.
7. Write code to get the Role Service from the policy domain (see Listing 6-18).

Listing 6-18 Get the Role Service Code

```
// Get the role service from the policy domain
       RoleService rmService = null;
       try {
            rmService = (RoleService) pd.getService(
                                                ServiceType.ROLE  );
       }
       catch( ServiceNotAvailableException naExc ) {
            // We could not fetch the service
            System.out.println( naExc.getLocalizedMessage() );
            System.exit( naExc.hashCode() );
       }
       System.out.println("Retrieved Role Service");
```

8. Optional: Write code to check the service type to make sure that it is a Role Service. If there are multiple versions of the Role Service available, you may also write code to check the version of the Role Service and verify that the version is compatible with the Role Service application. To program this step, use the code fragment shown in Listing 6-6, “Authentication Service Type, Version, and isCompatible Code,” on page 6-13 and substitute Role for Authentication wherever references to the AuthenticationService occur.

9. Write code to determine the roles of the authenticated user that is accessing the designated URL (see Listing 6-19).

Listing 6-19 Get Roles Code

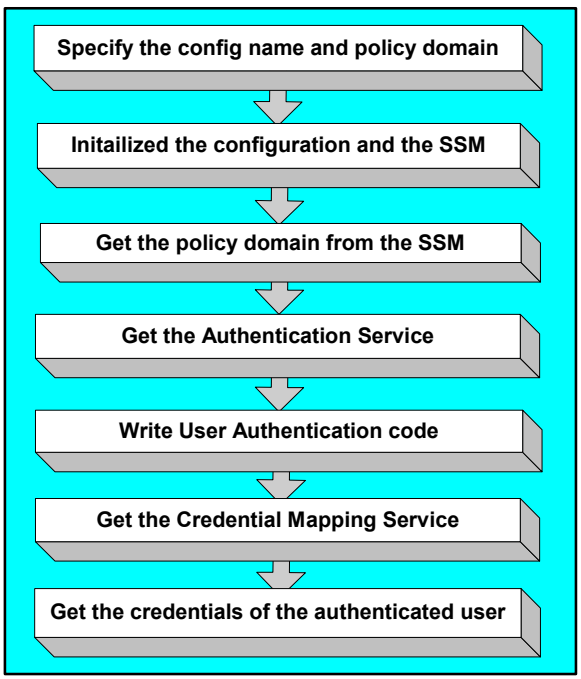
```
// Determine what roles myUser is in when accessing
// http://www.bea.com/index.html with a HTTP GET.
// Create the resource.
RuntimeResource beaurl = new RuntimeResource(
               "http://www.bea.com/index.html" , "URL_AUTHORITY" );

// Create the action.
RuntimeAction action = new RuntimeAction( "GET", "SIMPLE_ACTION" );

// Get a reference to the application context
AppContext ctx = MySession.getContext();

// Try fetching our roles
Vector roles=null;
try {
    roles = rmService.getRoles( myUser, beaurl,  action, ctx );
}
catch( IdentityNotAuthenticException identExc ) {
  // Exception is thrown if myUser is not properly authenticated
    System.err.println( identExc.getMessage() );
}

if ( roles != null ) {
  // Print the roles and descriptions
    for ( int loop=0; loop < roles.size(); loop++ ) {
       IdentityRole role = (IdentityRole) roles.elementAt( loop );
        System.out.println( role.getName() );
        System.out.println( role.getDescription() );
    }
}
```

Writing a Credential Mapping Application

You use the credential mapping service to allow the application to fetch credentials of certain types associated with a specific identity for a specific resource. These credentials can then be used on behalf of that identity to provide some privileged function, such as to logging into a database, or sending mail.

To write a Credential Mapping Service application, perform the following steps (see Figure 6-6):

Figure 6-6 Developing a Java Security Service Module Credential Mapping Application



1. Write code to specify the configuration name and the policy domain. The configuration is created and named in the Administration Application. Note that a policy domain shares the same name as the configuration used to create it. For a code fragment that shows how to program this step, see Listing 6-2, “Policy Domain Specification Code,” on page 6-11.
2. Write initialization code to initialize the configuration of the application and to initialize the Java Security Service Module to get an instance of it. For a code fragment that shows how to

program this step, see Listing 6-3, “Application Configuration and Java Security Service Module Initialization Code,” on page 6-12.

3. Write code to get the policy domain from the Java Security Service Module. For a code fragment that shows how to program this step, see Listing 6-4, “Get Policy Domain Code,” on page 6-12.
4. Write code to get the Authentication Service from the policy domain. For a code fragment that shows how to program this step, see Listing 6-5, “Get the Authentication Service Code,” on page 6-13.
5. Optional: Write code to check the service type to make sure that it is an Authentication Service. If there are multiple versions of the Authentication Service available, you may also write code to check the version of the Authentication Service and to verify that the version is compatible with the authorization application. For a code fragment that shows how to program this step, see Listing 6-6, “Authentication Service Type, Version, and isCompatible Code,” on page 6-13.
6. Write code to authenticate the user. For a code fragment that shows how to program this step, see Listing 6-7, “User Authentication Code,” on page 6-15.
7. Write code to get the CredentailMapping Service from the policy domain (see Listing 6-20).

Listing 6-20 Get the CredentialMapping Service Code

```
// Get the CredentialMapping service from the policy domain
       RoleService rmService = null;
       try {
            cmService = (CredentialMappingService) pd.getService(
                                          ServiceType.CredentialMapping  );
       }
       catch( ServiceNotAvailableException naExc ) {
            // We could not fetch the service
            System.out.println( naExc.getLocalizedMessage() );
            System.exit( naExc.hashCode() );
       }
       System.out.println("Retrieved CredentialMapping Service");
```

8. Optional: Write code to check the service type to make sure that it is a CredentialMapping Service. If there are multiple versions of the CredentialMapping Service available, you may also write code to check the version of the CredentialMapping Service and verify that the version is compatible with the CredentialMapping Service application. To program this step, use the code fragment shown in Listing 6-6, "Authentication Service Type, Version, and isCompatible Code," on page 6-13 and substitute CredentialMapping for Authentication wherever references to the AuthenticationService occur.
9. Write code to retrieve the credentials for the authenticate user that is accessing the designated URL (see Listing 6-21).

Listing 6-21 Get Credentials Code

```
// Retrieve credentials for MyUser according to the current resource and
// action being accessed and from a list of credential types that I am
// interrested in.
    // Create the resource.
    RuntimeResource beaurl = new RuntimeResource(
               "http://www.bea.com/index.html" , "URL_AUTHORITY" );

    // Create the action.
    RuntimeAction action = new RuntimeAction( "GET", "SIMPLE_ACTION" );

    // Try fetching the credentials.
    String[] askingCreds = new String[2];
    askingCreds[0] = "oracledb";
    askingCreds[1] = "3desDecryptionKey";
    Vector creds=null;
    try {
      // Both byIdent and forIdent are the same since I am fetching these
      // credentials for myself.
      creds = cmService.getCredentials( myUser, myUser, beaurl,  action,
                                        askingCreds );
    }
    catch( IdentityNotAuthenticException identExc ) {
        assertNull( identExc.getMessage(), identExc );
    }
    catch( ParameterException pExc ) {
```

```
    assertNull( pExc.getMessage(), pExc );
}

if ( creds != null ) {
   System.out.print("Returned " + String.valueOf( creds.size() ) +
                                "credential objects.");
    for ( int loop=0; loop < creds.size(); loop++ ) {
        Object cred = creds.elementAt( loop );

    // The type of objects returned depend on the credential type.
    // Either javax.resource.spi.security.PasswordCredential or
    // javax.resource.spi.security.GenericCredential are
    // recommended.
    // However, custom credential types are allowed.
        if ( cred instanceof
                javax.resource.spi.security.PasswordCredential ) {
          // This is a password credential.
      ...
    } else if ( cred instanceof
                javax.resource.spi.security.GenericCredential ) {
        // This is a generic credential.
          ...
    } else {
        // This is a custom credential type.
          ...
    }
    }
}
```