

Security Guide

JEUS 9

TMAXSOFT

Copyright

Copyright 2024. TmaxSoft Co., Ltd. All Rights Reserved.

Company Information

TmaxSoft Co., Ltd.

TmaxTower 8-9F, 29, Hwangsaеul-ro 258 beon-gil, Bundang-gu, Seongnam-si, Gyeonggi-do, South Korea

Website: <https://www.tmaxsoft.com/en/>

Restricted Rights Legend

All TmaxSoft Software (JEUS®) and documents are protected by copyright laws and international convention. TmaxSoft software and documents are made available under the terms of the TmaxSoft License Agreement and this document may only be distributed or copied in accordance with the terms of this agreement. No part of this document may be transmitted, copied, deployed, or reproduced in any form or by any means, electronic, mechanical, or optical, without the prior written consent of TmaxSoft Co., Ltd. Nothing in this software document and agreement constitutes a transfer of intellectual property rights regardless of whether or not such rights are registered) or any rights to TmaxSoft trademarks, logos, or any other brand features.

This document is for information purposes only. The company assumes no direct or indirect responsibilities for the contents of this document, and does not guarantee that the information contained in this document satisfies certain legal or commercial conditions. The information contained in this document is subject to change without prior notice due to product upgrades or updates. The company assumes no liability for any errors in this document.

Trademarks

JEUS® is registered trademark of TmaxSoft Co., Ltd.

Java, Solaris are registered trademarks of Oracle Corporation and its subsidiaries and affiliates.

Microsoft, Windows, Windows NT are registered trademarks or trademarks of Microsoft Corporation.

HP-UX is a registered trademark of Hewlett Packard Enterprise Company.

AIX is a registered trademark of International Business Machines Corporation.

UNIX is a registered trademark of X/Open Company, Ltd.

Linux is a registered trademark of Linus Torvalds.

Noto is a trademark of Google Inc. Noto fonts are open source. All Noto fonts are published under the SIL Open Font License, Version 1.1. (<https://www.google.com/get/noto/>)

Other products and company names are trademarks or registered trademarks of their respective owners.

The names of companies, systems, and products mentioned in this manual may not necessarily be indicated with a trademark symbol (™, ®).

Open Source Software Notice

Some modules or files of this product are subject to the terms of the following licenses: APACHE2.0, CDDL1.0, EDL1.0, OPEN SYMPHONY SOFTWARE1.1, TRILEAD-SSH2, Bouncy Castle, BSD, MIT, SIL OPEN FONT1.1

Detailed Information related to the license can be found in the following directory:
\${INSTALL_PATH}/license/oss_licenses

Document History

Product Version	Guide Version	Date	Remarks
JEUS 9	3.1.1	2024-12-24	-

Contents

1. Introduction to the Security System	1
1.1. Overview	1
1.2. Key Features	2
1.3. Architecture	2
1.4. Core Concepts	4
1.4.1. Login	4
1.4.2. Authentication	5
1.4.3. Authorization	6
1.4.4. Auditing	12
1.4.5. Services and SPI	12
1.4.6. Domain	14
1.5. Improving Performance and Security Level	16
1.5.1. Tuning the Security System	16
1.5.2. Improving Security Level	16
2. Configuring the Security System	19
2.1. Overview	19
2.2. Configuring the Security System Domain	20
2.2.1. Configuring XML	20
2.2.2. Configuring User Accounts and Security Policies	22
2.3. Configuring Security Domain Components	23
2.3.1. Configuring XML	23
2.4. Configuring Security Services	25
2.4.1. Configuring XML	25
2.5. Configuring the Security System User Information	30
2.5.1. Configuring XML	30
2.5.2. Using Database	32
2.5.3. Configuring Password Security	33
2.5.4. Cached Login Information	38
2.6. Configuring Security System Policies	38
2.6.1. Configuring XML	39
2.6.2. Using Database	44
2.7. Configuring Additional Settings	45
2.7.1. Configuring Java SE SecurityManager	46
2.7.2. Configuring JACC Provider	47
2.7.3. Configuring information to Grant Identity	47
2.7.4. Configuring Identity Certificate Information	48
3. Configuring Security in Applications and Modules	50
3.1. Overview	50

3.1.1. Module Deployment vs. Application Deployment	50
3.1.2. Role-to-Resource Mapping	50
3.1.3. Principal-to-Role Mapping	52
3.1.4. User Configurations	54
3.2. Configuring EJB Module Security	55
3.2.1. Configuring ejb-jar.xml	55
3.2.2. Configuring jeus-ejb-dd.xml	58
3.3. Configuring Web Module Security	61
3.3.1. Configuring web.xml	61
3.3.2. Configuring jeus-web-dd.xml	65
3.4. Configuring Jakarta EE Application Security	67
3.4.1. Configuring application.xml	67
3.4.2. Configuring jeus-application-dd.xml	68
3.5. Example	70
4. Programming with the Security System API	72
4.1. Overview	72
4.2. Configuring Java SE Permissions	72
4.3. Basic API	72
4.4. Resource API	73
4.5. SPI Class	74
4.6. Example	75
5. Developing Customized Security Services	77
5.1. Overview	77
5.2. Service Class	77
5.3. The Basic Pattern of Implementing Custom Security Services	79
5.4. SPI Class	80
5.4.1. SubjectValidationService SPI	81
5.4.2. SubjectFactoryService SPI	82
5.4.3. AuthenticationService SPI	82
5.4.4. AuthenticationRepositoryService SPI	82
5.4.5. IdentityAssertionService SPI	83
5.4.6. CredentialMappingService SPI	83
5.4.7. CredentialVerificationService SPI	84
5.4.8. AuthorizationService SPI	84
5.4.9. AuthorizationRepositoryService SPI	85
5.4.10. EventHandlingService SPI	85
5.4.11. Dependencies between SPI Implementations	86
5.5. Security Services Configurations	86
6. Using JACC Provider	87
6.1. Overview	87
6.2. Introducing JACC Protocol	87

6.2.1. Provider Configuration Protocol	87
6.2.2. Policy Configuration Protocol	88
6.2.3. Policy Decision and Execution Protocol	89
6.3. Developing JACC Provider	89
6.3.1. Implementing JACC Provider	90
6.3.2. Packaging JACC Provider	91
6.3.3. Default JACC Provider	91
6.4. Integrating JACC Providers with the JEUS Security System	92
7. Using JAAS	96
7.1. Overview	96
7.2. Implementing LoginModule to Integrate JEUS with LDAP	97
7.3. Configuring LDAP JAAS LoginModule Service	101
7.4. Implementing LoginModule to Integrate with Database	103
7.5. Configuring LoginModule Service	109
Appendix A: Security Event Service	112
A.1. Overview	112
A.2. Event	112
Appendix B: JEUS Server Permissions	116
B.1. Overview	116
B.2. JEUS System Resource Name	116
B.3. jeusadmin Command Permission Configurations	117
References	119

1. Introduction to the Security System

This chapter describes the concepts, structure, and key features of the security system.

1.1. Overview

In general, the concept of security refers to a set of measures used for preventing and detecting potential and actual infringements on a system that might cause internal and external damage.

This concept is relevant to an enterprise or computing environment that manages user information and provides users with services. In this environment, security measures are provided to protect the users' names, passwords, and addresses as well as resources that provide a service or operate the system. To protect the critical data and system resources, each enterprise or computing system uses software or hardware designed to provide security.

JEUS provides various security services as well. The JEUS security services implement Service Provider Interface (SPI), which provides various security service types that protect user information and resources defined in JEUS.

This guide outlines the basic structure of the JEUS security system and general security management.

The JEUS security system is designed to achieve the following:

- Have a flexible and pluggable framework.

The framework should be able to integrate the security system with any existing third-party security mechanism and/or persistent storage mechanism.

- Ensure security.

The security system should prevent any unauthorized access to the system, even by a highly sophisticated intruder who can decompile the security system sources.

- Provide high performance.

The security system should have minimal impact on the overall performance while ensuring security for the entire system and application codes. While there is a general tradeoff between security level and system performance, the JEUS security system is designed to offer both maximum security and high performance.

- Provide a simple, unambiguous API and SPI for facilitating maintenance and the creation of 3rd party security services.
- Maintain data integrity.
- Comply with the standards.

The JEUS security system meets all the mentioned objectives.

1.2. Key Features

The following are the key features of the JEUS security system:

- The system has an open architecture and framework with a complete set of security integration SPI for customers and 3rd parties to use as needed.
- The system fully supports dynamic principal-to-role and role-to-resource authorization mappings.

This means that the authorization mappings (principal-to-role and role-to-resource) are applied at runtime.

This enables the following security policies:

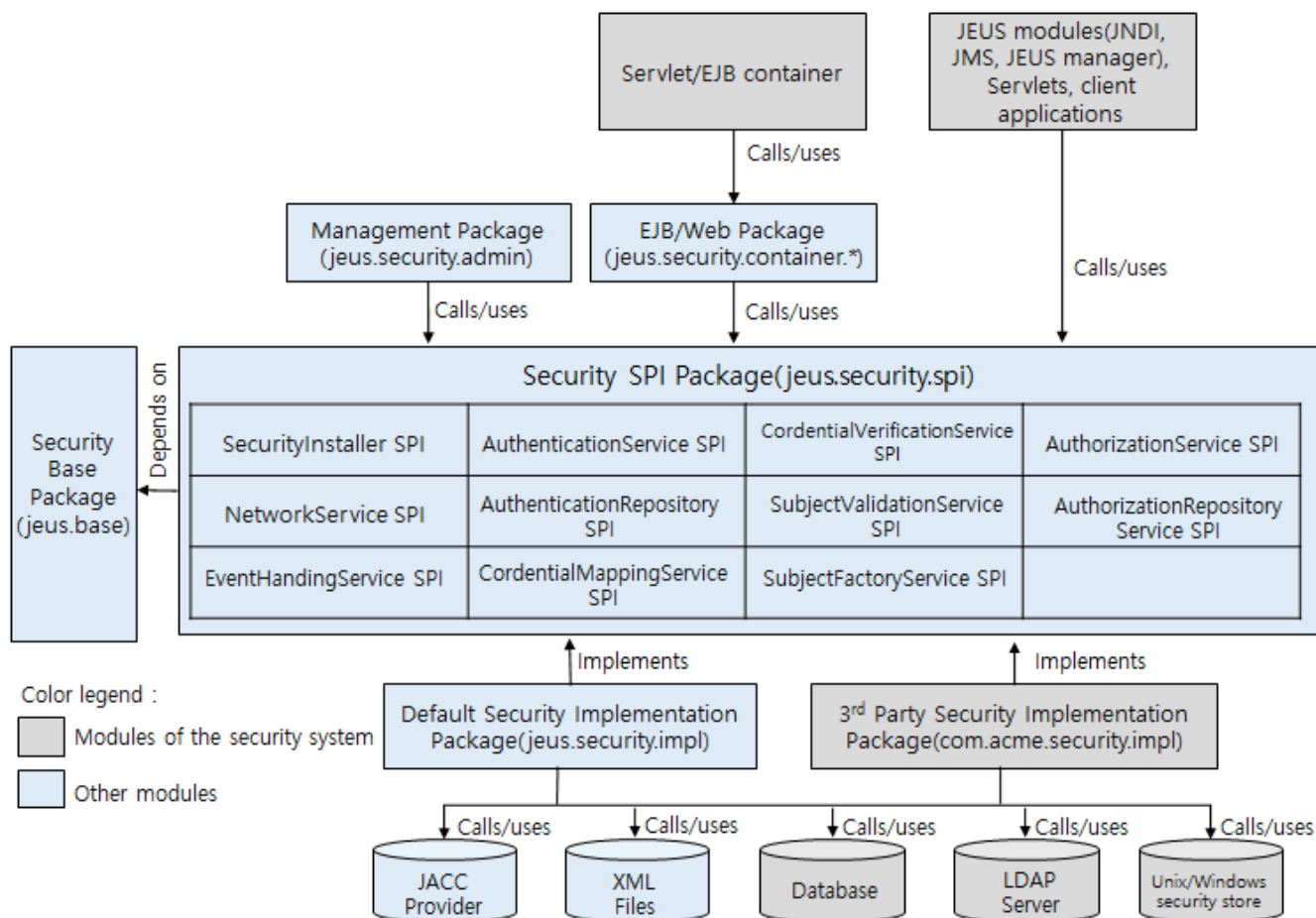
- User U is granted the R role only during the 9 to 5 business hours, Monday to Friday
- Everyone is granted the R role
- No-one is granted the R role
- The system supports the concept of a security domain (hereafter domain). This allows different Jakarta EE applications access different security services.

Note that domain and security domain refer to different concepts in the JEUS system. In the system, a domain refers to a server management unit and a security domain refers to a security management unit.

- The system provides default implementations of all critical security operations such as authentication, authorization, repositories, auditing, and clustering.
- The system supports security auditing mechanisms through a flexible event handling model.
- The system enables the user to add simple security functions, such as Subjects and policies, to the servlet, EJB, and application source codes.
- The system fully supports Java SE 8 and JACC 2.0 specifications.
- The system provides full documentation support through this document and additional Javadoc.
- The system runs independently from other JEUS modules, which makes it easy to use in a different context.

1.3. Architecture

The following figure shows the basic architecture of the security system.



Architecture of the Security System

The following are the main components (package) of the security system:

- jeus.security.spi

The security SPI classes (represented by the large box in the middle of the figure) is the core of the JEUS security system. These abstract classes contain methods that are called by user code (JEUS containers etc.), as well as abstract methods that will be implemented by third party sub-classes.

There are currently eleven SPI classes available, each related to a specific aspect of security such as authentication and authorization.

- jeus.security.base

The security base package (represented by the small box in the left) includes basic, concrete implementations and interfaces that are referenced by the security SPI classes. This package includes two important classes, Subject and Policy.

- jeus.security.impl.*

The default security implementation package (represented by the box at the left bottom corner) currently supports XML file repositories and JACC providers.

- jeus.security.admin, jeus.security.container

The management package and the EJB/Web container package (represented by boxes right above the center) contain codes that use the SPI classes.

- Repository and external security mechanism

Repositories and external security mechanisms.

They (represented by the boxes at the bottom) include databases, LDAP servers, or XML files. The repositories are used to persistently store security attributes, such as Subject and Policy information.

External security mechanisms refer to mechanisms that perform authentication or authorization. An example of an external security mechanism is the JACC provider. However, the boundaries for plain repositories and security mechanisms are not clearly defined. As shown, it is the responsibility of the SPI implementation classes (in this case, the class included in `jeus.security.impl.*` package) to decide which repository and security mechanism to apply.

1.4. Core Concepts

This chapter covers the core concepts necessary to understand the overall security system.

- [Login](#)
- [Authentication](#)
- [Authorization](#)
- [Auditing](#)
- [Service and SPI](#)
- [Domain](#)

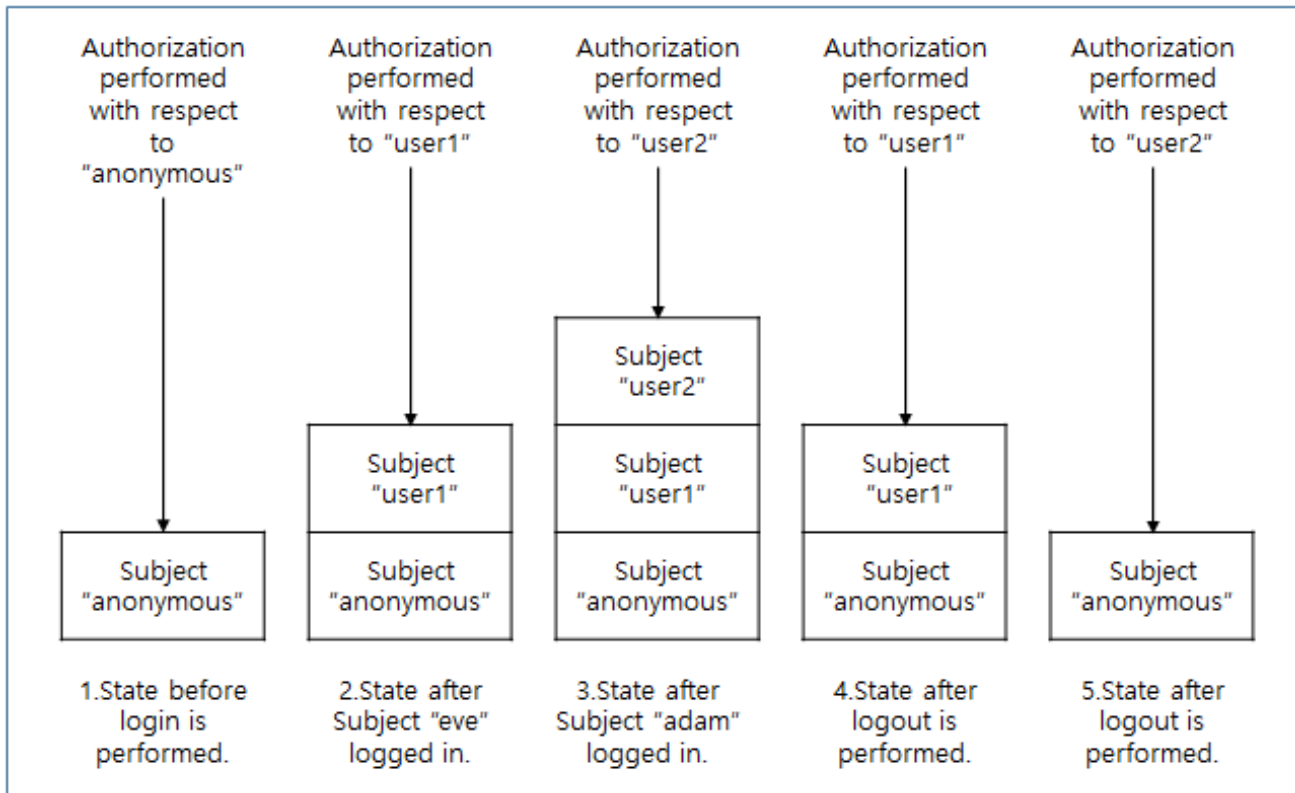
1.4.1. Login

In the JEUS security system, a login is a process that associates a Subject with a Java execution thread. This process is different from authentication, which takes place before the Subject is associated with an execution thread. Therefore, if the authentication is successful, a login will proceed. But if the authentication fails, a login also fails.

After a Subject has logged in successfully, the Subject goes through authorization. The login concept may thus be seen as an umbrella that spans both authentication and authorization. The opposite of a login process is a logout process, which disassociates a Subject from its execution thread.

Note that the implementation of a login process requires a stack-based operation. Several Subjects can be logged in at any time with each new Subject being pushed onto the top of a stack. Only the Subject on top of the stack is used for authorization. When the Subject logs out, the top Subject is removed from the stack, and the next Subject at the top of the stack is activated.

The following illustrates this mechanism.



Stack-based Login Mechanism

In the JEUS security system, only one login stack is allowed for each Java thread. As shown in the Security System Architecture figure, the login mechanism is implemented by the `jeus.security.impl.login.CommonLoginService` interface.

1.4.2. Authentication

Authentication is a process of obtaining the identity of a caller for the purpose of using it for authorization at a later time.

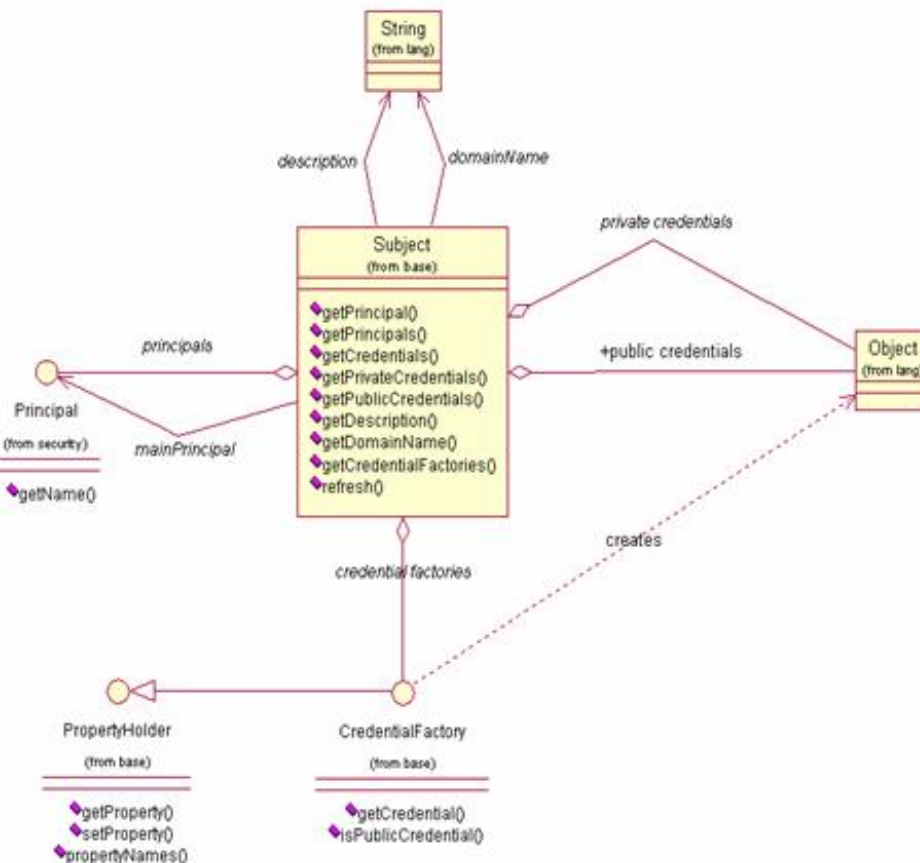
In the JEUS security system, an identity is represented as a Principal that is defined by the `java.security.Principal` interface.

A Principal is stored as an attribute value of a Subject, which, as described in the previous section, is implemented by the `jeus.security.base.Subject` class and associated with an execution thread. A Subject includes Principals and credentials as its security attributes.



The Subject implementation used in the JEUS security system is not equivalent to the JAAS Subject defined by the class `javax.security.auth.Subject`. However, these two implementations are similar in many aspects. Also, it is possible to convert from one to the other so that even if some information is lost, that lost information can be referred to by an alias.

The following is a UML diagram for a JEUS Subject.



Subject UML Diagram

As shown, the Subject has a main Principal as its unique ID. The Subject may also carry any number of additional optional Principals. These are non-unique Principals that may be shared by several Subjects, and these non-unique Principals are referred to as group group Principals.

Subject can also have public or private credentials. Credentials are usually used as a proof to authenticate the Subject or to deliver specific information. An example of a private credential is a password. An example of a public credential is a digital certificate.

The credentials of a given Subject may be created by using its sub-class, credentialFactory class. The actual credentials are obtained from the credential factories using the `refresh()` method, and they are added to the public or private credential sets.



Each Subject always belongs to exactly one domain. A Subject with the main Principal "user1" in domain A is thus different from another Subject with the main principal "user1" in domain B.

1.4.3. Authorization

In the JEUS security system, authorization is a process of determining whether a previously authenticated Subject should be allowed to perform a particular operation.

Authorization happens at the system level. For example, authorization is required to determine whether a certain Subject, usually an administrator, is authorized to boot or shut down the JEUS server. Authorization is also required at the application level, for example, when a JEUS engine checks whether a remote caller should be allowed to access a particular application component, such as a specific EJB method or a servlet.

As in Jakarta EE, authorization in the JEUS security system is a role-based mechanism. The developer or assembler of a Jakarta EE application sets up security restrictions that are assigned to roles, to which Principals are mapped during the deployment of an application.

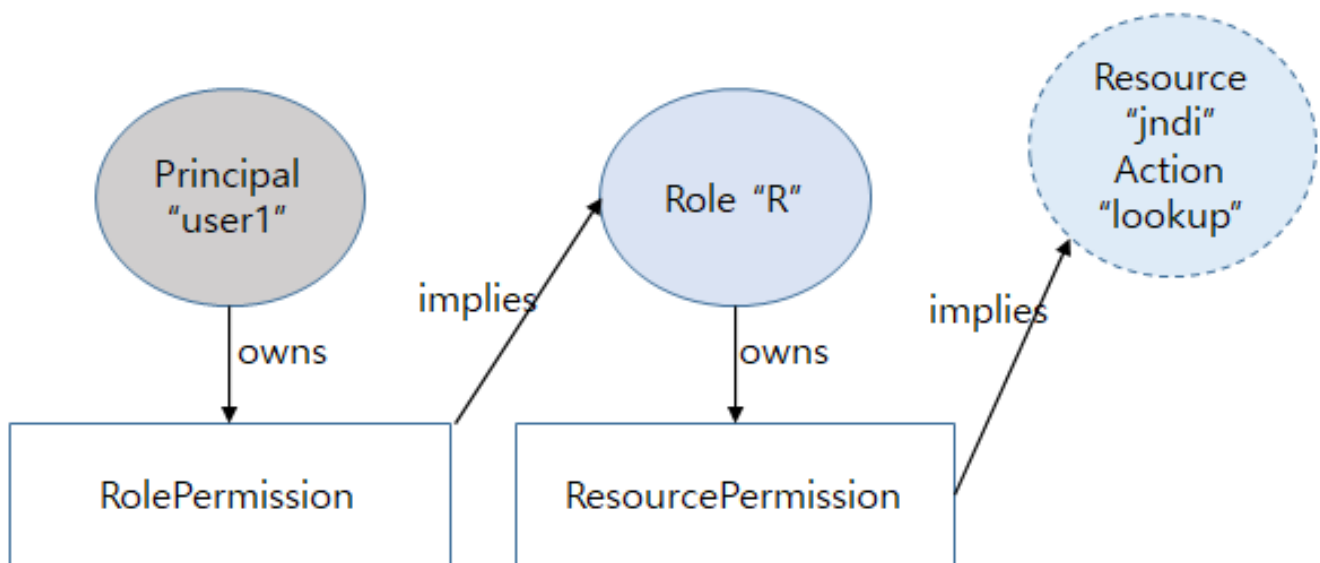


The role-based approach is also used for JEUS system authorization as well as for Jakarta EE applications.

In the JEUS security system, authorization mappings are referred to as permissions (sub-classes of `java.security.Permission`). For example, the Principal "user1" is said to have the permission to access a role called R. That is, user1 belongs to the role group R.

Let's suppose that the role R includes the permission to access the resource JNDI and execute the lookup action.

The following illustrates role-based authorization.



Role-based Permission Authorization

As seen in the previous figure, only Principals and roles are represented as physical entities. The resource circle in the figure has a dashed outline to denote that resources are implied and not physically modeled in the authorization process.

In the previous figure, the two boxes marked RolePermission and ResourcePermission represent instances of the two classes `jeus.security.resource.RolePermission` and `jeus.security.resource.ResourcePermission`, respectively. Both of these classes extend the basic `java.security.Permission` abstract class, and there are various other sub-classes of the Permission class.

Briefly summarized, a Principal owns a set of permissions and those permissions grant the Principal a set of logical roles. These roles in turn own another set of permissions that grant those roles access to a set of actions on the resources. By following these indirect mappings, one can determine whether a principal should be allowed to perform a specific action on a resource.

The figure above shows that the Principal user1 owns a RolePermission. The arrows marked "owns" in the figure intuitively denote a direct, static binding. However, the term "implies" used to show that the RolePermission implies role R is different. This implication indicates that binding is not static. Rather, the relationship between the RolePermission and the role R is determined dynamically at runtime by invoking the method `implies(Permission p)` on then RolePermission instance.

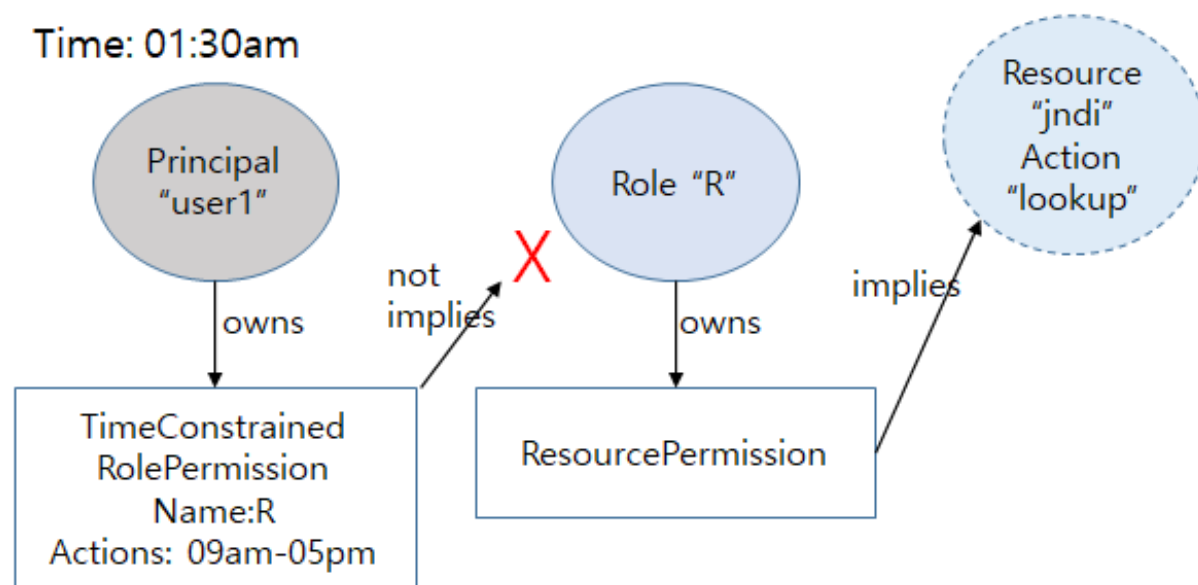
If the `implies()` method returns true, the RolePermission is said to imply role R, otherwise it is not. Thus, in the former case, principal user1 is granted the role R, and in the latter case it is not. The same logic applies for the implication between the role R and the Resource JNDI.



For more information about Permission, refer to the `java.security.Permission` section in Java SE JavaDoc.

Given the previous information, it is not difficult to implement dynamic mappings. If we just change the implementation of the `implies()` method, we can make a permission imply a role or resource under certain conditions. For example, we might want to create a dynamic mapping between principal user1 and role R, so that user1 is granted with role R during the business hours (9 AM to 5 PM). To do this, we could simply create a new RolePermission sub-class, called TimeConstrainedRolePermisson, and override the implementation of the `implies()` method. In this method, you can add a check for the additional time-constraint.

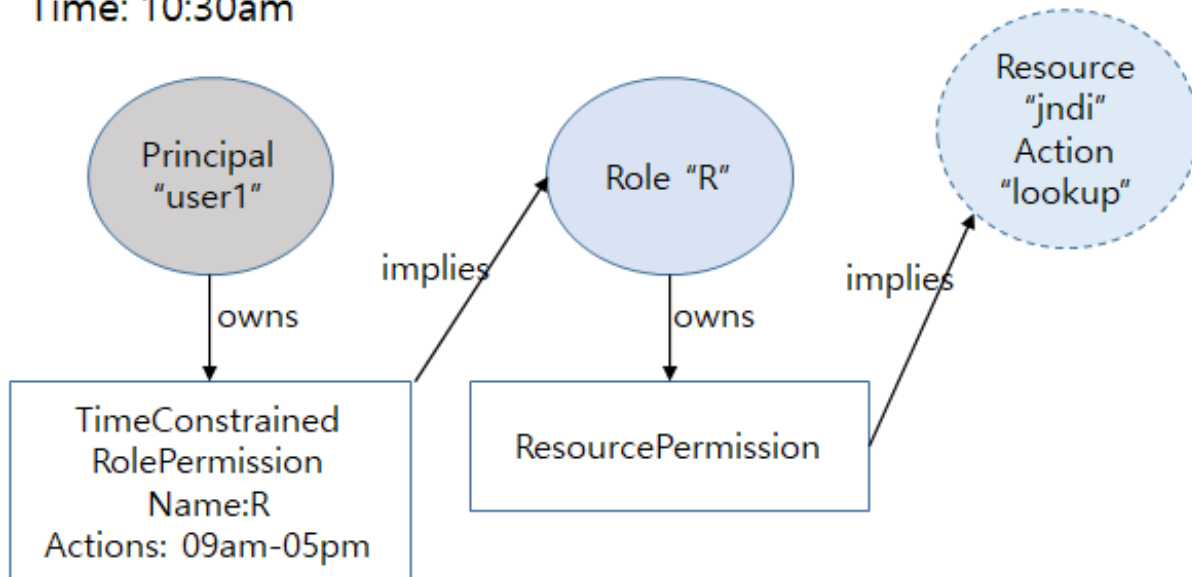
For more examples of implementing dynamic mappings, refer to [Role at 01:30 AM](#) and [Role at 10:30 AM](#).



Role at 01:30 AM

As shown in [Role at 10:30 AM](#), principal user1 is not granted role R.

Time: 10:30am



Role at 10:30 AM

As shown in the previous figures, the TimeConstrainedRolePermission class contains two variables, Name and Actions. Name is set to the name of the implied role, which is R in the example, and the Actions are set to the time when the implication is valid, which is between 9 AM and 5 PM in the example. Both Name and Actions are standard variables that are declared in most java.security.Permission class implementations.

Moreover, apart from the basic Permission-based mappings, every permission in the authorization system belongs to one of the following three types.

Classification	Description
Excluded Permission	<p>No one can have authority for this permission.</p> <p>For example, an excluded permission, that grants access to the resource RSC, will not allow anyone to access RSC.</p> <p>Excluded Permissions have higher priority than unchecked Permissions, in case there is an overlap.</p>
Unchecked Permission	<p>Permission owned by all users.</p> <p>For example, an unchecked permission that grants access to a resource "RSC will allow anyone to access RSC, regardless of his or her role(s).</p> <p>Unchecked Permissions have lower priority than excluded Permissions but higher priority than checked Permissions, in case there is an overlap.</p>
Checked Permission	<p>Permission owned by a Principal or a role. Examples of these Permissions are illustrated in the previous 2 figures.</p>

Several examples for this will be further discussed later.

In the JEUS security system, all permission mappings are the classes that implement the jeus.security.base.PermissionMap class. These PermissionMap classes are included in the

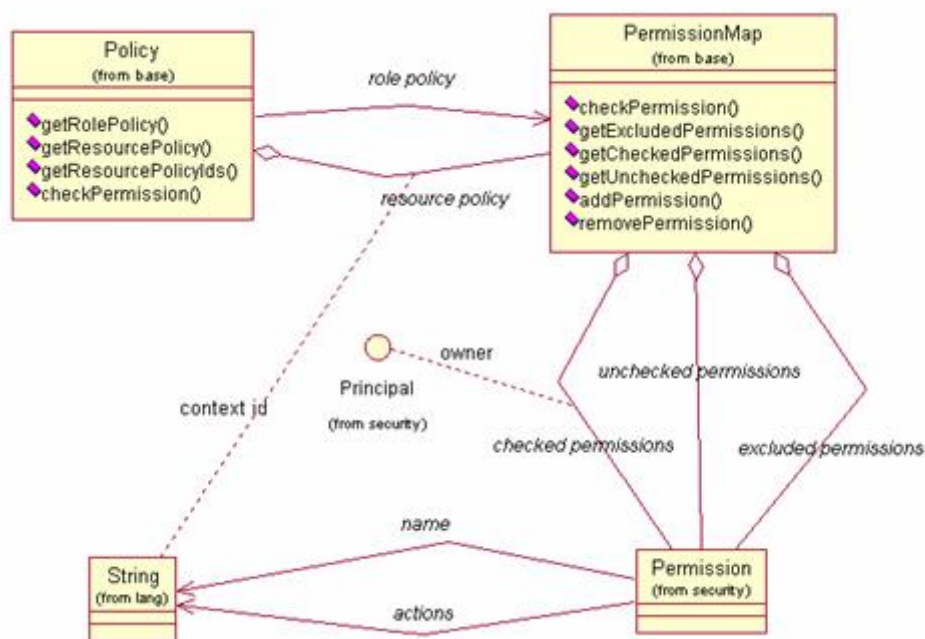
jeus.security.base.Policy class.

Policy class has two kinds of PermissionMaps.

- Principal-to-Role Map (Role Policy0)
- Role-to-Resource Map (Resource Policy)

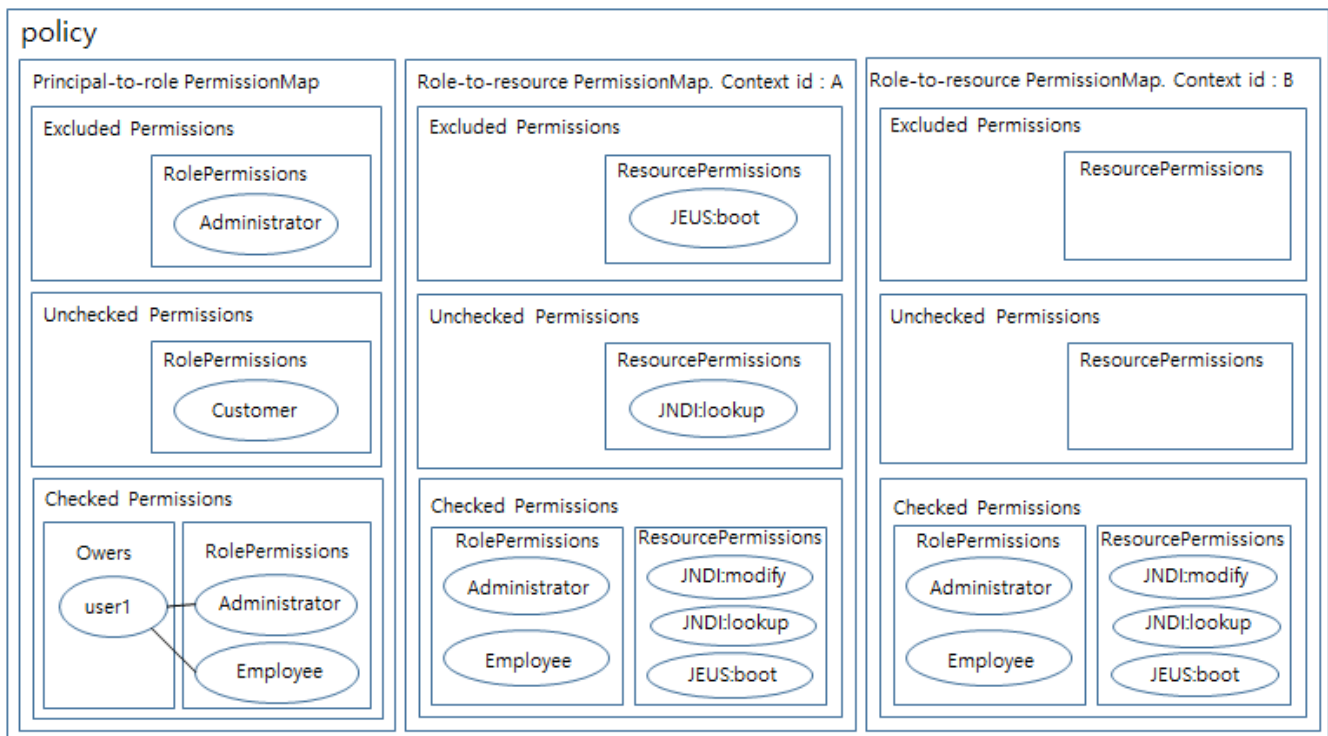
Each PermissionMap contains the three aforementioned permission types—excluded, unchecked and checked permissions. In the case of checked permissions, a permission and role PermissionMap are combined through a Principal or role. A Policy and resource PermissionMap are combined through a context ID, which indicates the authorization scope.

The following is a UML diagram of Policy and PermissionMap classes.



UML Diagram of Policy and PermissionMap

The following figure summarizes information about mappings.



Example of Policy with One Principal-to-Role mapping and Two Role-to-Resource mappings

In the previous figure, we have a Policy instance that contains one principal-to-role mapping, which is required, and also two role-to-resource mappings with context IDs A and B, respectively. All three PermissionMaps contain three sets of permissions as follows.

- Excluded Permissions
- Unchecked Permissions
- Checked Permissions

The oval figures inside the boxes represent the actual permission instances (Name: Action) with Name and Action variables.

Let's suppose that a Subject, with context ID "A" and the Principal "user1", wants to access JNDI to modify it.

This authorization query would be processed by the authorization system as follows:

1. A new ResourcePermission (hereafter RSP), is created with the name "JNDI" and action "modify."
2. The RSP is passed along with the context ID "A" and Principal "user1" to the Policy.
3. Policy selects the Role-to-ResourcePermissionMap for context ID "A."
4. Policy checks whether excluded permissions of PermissionMap A contain the RSP, and receives the result that it is not included in excluded permissions.
5. Policy checks whether unchecked permissions of PermissionMap A contains the RSP, and receives the result that it is not included in unchecked permissions.
6. Policy checks whether checked permission of PermissionMap A contains the RSP, and receives the result that it is included in Checked Permission.
7. Policy retrieves a list of permission owners that implied the RSP, and finds that there is only one

such owner, the role called Administrator.

8. Policy constructs a RolePermission (hereafter RLP) object with the name Administrator.
9. Policy retrieves the Principal-to-RolePermissionMap.
10. Policy checks whether excluded permission of Principal-to-RolePermission contains RLP, and receives the result that RLP is included in Excluded Permission.
11. The fact RLP is included in excluded permission indicates that the role Administrator has excluded permission. Therefore, nobody can access Policy. As a result, the entire authorization process will be terminated, and the value DENIED will be returned.

In this process, although user1 is mapped to the role Administrator, since Administrator is also in the excluded set, user1 is not granted the role Administrator. This is because excluded permissions have higher precedence over both unchecked and checked permissions.

If you follow the instructions explained above, you will be able to see how the following authority check query is solved.

Principal	Operation	Context	Outcome
user1	JNDI:lookup	A	GRANTED
user1	JNDI:modify	A	DENIED
user1	JEUS:boot	A	DENIED
user1	JEUS:boot	B	GRANTED
Anonymous	JNDI:lookup	A	GRANTED
Anonymous	JNDI:lookup	B	DENIED
Anonymous	JEUS:boot	A	DENIED
Anonymous	JEUS:boot	B	DENIED

1.4.4. Auditing

Security auditing generally involves capturing security-related events such as failed authentication and runtime exceptions and evaluating these events for better system security.

The JEUS security system features a simple yet flexible auditing mechanism using security events. Whenever something significant happens in the system, such as failed authentication or failed authorization, an event will be posted to a set of registered event handlers.

The handler implementations, which can be fully customized, responds with an appropriate action, such as locking a Subject's credential when authentication has failed too many times in a row.

1.4.5. Services and SPI

Each class in the security system that implements security functionalities constitutes as a Service. A Service is simply an implementation of an Service Provider Interface (SPI) that provides some security

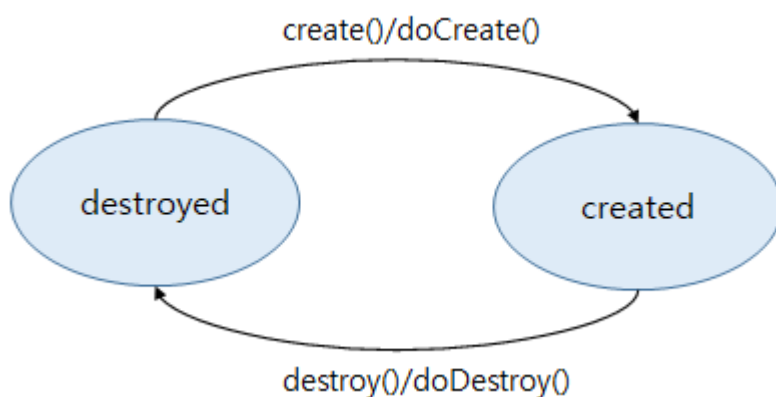
functionality, whether it is authentication, authorization, or networking.

SPI is an abstract class of the Java package `jeus.security.spi`. These SPI class should be extended and implemented in order to implement customized security functionality. The extended sub-classes of an SPI class are called Services.

All instances of Services in the security system are treated individually as independent entities whose services are called as needed. However, it is common for a Service implementation to make calls to another SPI, and some dependency may exist among different Services. In order to initialize Services. All Services receives a key-value pair property value used to initialize the Service. The configuration data of a Service can also be saved in a configuration file.

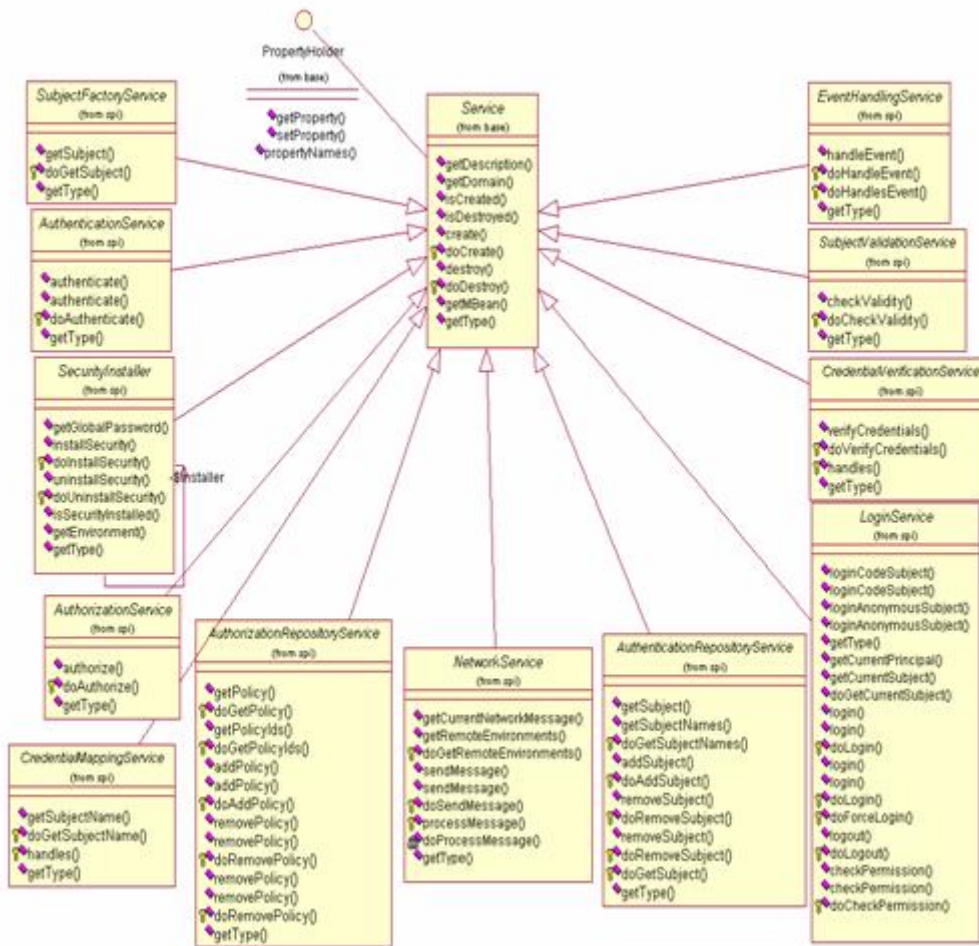
All Services may also optionally define a JMX bean that will be exported to the JEUS management system. The JMX MBean is normally created based on a MBeanInfo instance returned by the `Service.getMBeanInfo()` method.

All Services have two states, a created state and a destroyed state. The `create()` and `destroy()` methods are called to transition between the two states. Calling these methods will invoke `doCreate()` and `doDestroy()` abstract methods. These abstract methods should be implemented by Service sub-classes for service initialization and management.



Two Types of Service

The following shows a diagram with a variety of service classes and SPI classes included in the `jeus.security.spi` package.



Service Classes and SPI Sub-classes

At runtime, service instances are created by the singleton class called SecurityInstaller. A simple SecurityInstaller class can be implemented just by writing the necessary code. A more sophisticated implementation would save the service name and property values to a specific configuration file and use the information to instantiate and initialize the Service. The default SecurityInstaller implementation operates according to the latter approach, providing a very simple and flexible way to add new Service implementations to the security system.



Users do not need to understand the Service and SPI architecture when only using the JEUS security system.

1.4.6. Domain

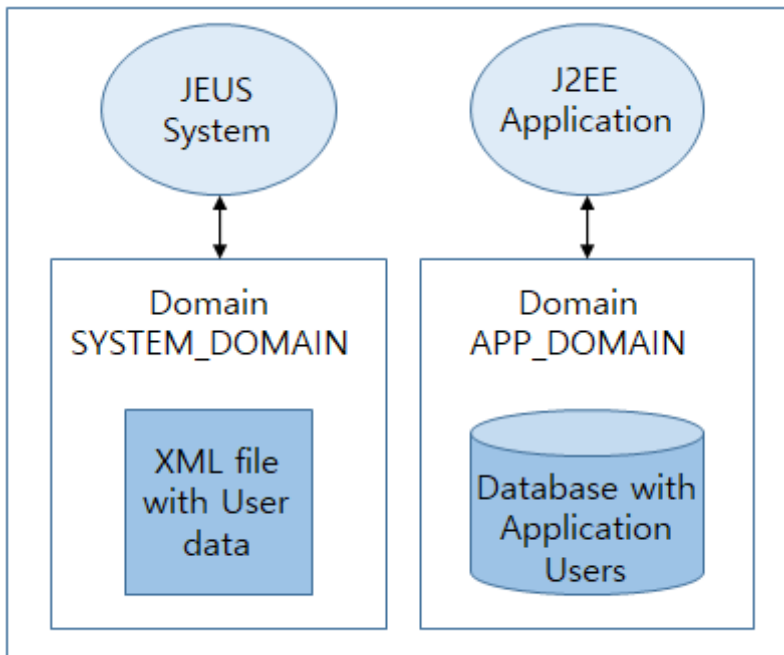
Security domain is a collection of security Service instances. Several domains may exist concurrently within the security system. The purpose of a domain is to allow deployed applications and JEUS sub-systems to use individual security Services. A domain separates the security Services for each application.

For example, the JEUS server could be set up to use a special domain. In this case, it is call the SYSTEM_DOMAIN. This domain contains Subjects and Policies that are used to manage the JEUS server. For example, the SYSTEM_DOMAIN contains the user information for a main Subject called

administrator, which can be used to boot or shut down the server.

Another domain, let's call it APP_DOMAIN, could be used by a deployed Jakarta EE application. That domain might also contain a user information for a Subject called administrator, which is different from the JEUS server administrator. You could also configure a different repository mechanism for each domain. While the SYSTEM_DOMAIN can be configured to use a simple XML file to store Subject information, the APP_DOMAIN can be configured to use a remote database to read and write Subject information.

The domain concept is illustrated in the following figure.



Two Domains using Different Application and Subject Repository

Use of the domain feature is optional. If no domain is specified, a special SYSTEM_DOMAIN is used.



It is our convention to name domains using capital letters and to append "_DOMAIN" to the name. This is not a requirement, just a convention. Also note that whenever the term domain is used within this document, it refers to the Security domain. These Security domains have nothing in common with Jakarta EE server domains.

By default, JEUS uses two domains, SYSTEM_DOMAIN and SHARED_DOMAIN, as follows:

- SYSTEM_DOMAIN

The domain name that is used when managing the JEUS server through its standard management tools. This domain contains JEUS administrator accounts and permissions to manage the JEUS server, such as booting and shutting down. By default, this domain is also used for Jakarta EE applications, and it can be changed through configuration.

- SHARED_DOMAIN

The name of a special domain whose security services will be shared by all other configured domains. Thus, the SHARED_DOMAIN contains security services that will be applied to all other domains.

To apply different security services to applications, create a separate domain and configure the domain.xml file. For more information about creating and configuring security domains, refer to [Configuring Security System](#).

1.5. Improving Performance and Security Level

To a certain degree, the information given in this chapter conflicts with the attributes a security system must have. Basically, there is always a tradeoff between performance and security. Thus, you need to control the level of performance and security to an appropriate level according to the task requirements.

1.5.1. Tuning the Security System

This chapter explains how to minimize the performance impact that the security system has on the overall performance of the JEUS server.

The following are measures for improving performance of the overall security system.

- **Do Not Use Secured Connection in JEUS Domain.**

Secure connection uses a socket to send an encoded packet, which means that larger amount of data will be transmitted than with a general socket communication.

To improve performance, it is recommended not to use secured connections in JEUS domain.

- **Do Not Use JEUS SecurityManager**

You can avoid executing unnecessary operations by not using the SecurityManager. This means that authorization is not used, but instead authentication is required.

- **Use Non-blocking I/O**

Network blocking occurrences can be reduced by using Non-Blocking I/O. Non-blocking I/O removes blocking while waiting for the next I/O request to occur.

Therefore, CPU usage and the number of FD are reduced because a separate thread for I/O is not needed for each connection. non-blocking I/O is used in JEUS by default. It guarantees higher performance than blocking I/O.

1.5.2. Improving Security Level

This section explains how to improve the security system level despite a performance hit.

The following are the ways to improve the overall security level of JEUS:

- **Set the global system password.**
- **Use secured connection in JEUS security domain**

Configure the network security service to use secured connection, which is usually protected with SSL/TLS. For more information about how to set a secured connection when the default network service is used, refer to [References](#). This prevents anyone from network eavesdropping and picking up sensitive information.

- **Protect Subject and Policy from Unauthorized Access**

It is essential to protect user information of a Subject and the Policy repository location from unauthorized access. The method for this depends on the repository type. The following are some example cases.

- If a database is in use, the table that stores user information of a Subject and the table where Policy is stored must be protected through authentication and authorization configured on the database side. You must also protect connections between the JEUS security repository and the database by using SSL, etc. Refer to the database documents to learn how to configure this setting.
- If security information is stored in an XML file such as accounts.xml and policies.xml, you must create proper file access permissions to ensure that only the JEUS security system and administrators have access to this file. Use encryption, if encryption is supported regardless of the repository. Refer to the repository document to learn how to apply security to files. Refer to the operating system manual to learn how to set read/write access to the files.

- **Use Security Auditing**

Create a security event log for events that occur in the system and periodically audit the log. JEUS security auditing mechanism is usually implemented by using the service SPI class, jeus.security.spi.EventHandling. Note that a log file must be protected from unauthorized access. For information about various security auditing mechanisms, refer to [References](#).

- **Use JavaSE SecurityManager**

Configure JavaSE SecurityManager as explained in [Configuring Java SE SecurityManager](#) to improve the robustness of the JEUS system and to protect JEUS from potentially malicious EJB and Servlet code injections.

- **Use Third-Party Security Mechanism**

In an environment that deals with sensitive information such as in a banking application, it is strongly recommended to use additional security mechanisms such as firewalls or virtual private networks (VPNs) along with the JEUS security system.

- **Configure Security for Operating System**

Regardless of the operating system in which JEUS operates, do as follows to protect the operating system.

- Restrict physical access to the servers where JEUS is installed. For example, keep the servers in a locked place.
- Set process and file privileges so that only trusted administrator can access the JEUS files and processes. Refer to the operating system manual.
- Update the operating system with the latest security related patches.
- Periodically execute an antivirus software.
- Keep all written security related documents safe. For example, keep written passwords in a locked cabinet.

2. Configuring the Security System

This chapter describes how to install and configure the security system in JEUS. For information about configuring other services in a domain other than what is mentioned here, refer to [References](#).

2.1. Overview

This section will briefly summarize some basic aspects related to security system configurations.

The JEUS security system starts when the server is executed by the Security Installer. To protect JEUS, the Security Installer creates a security domain based on the information in the domain.xml file before starting the security service. The user accounts and security policies defined in the account.xml and policies.xml files are applied in the default security system.

The following two items can be configured in a security domain.

- Definitions of security domain and security services.
- Accounts and policies for the domain.

The following are steps for configuring the default security system.

1. Configure security domains.
2. Configure security Services for each security Domain.
3. Configure Subjects (authentication data) for each Domain.
4. Configure Policies (authorization data) for each Domain.
5. Configure additional items.
6. Configure a JavaSE SecurityManager (optional).
7. Configure a JACC provider (optional).

Default Security System Directory

The following is the directory structure of the default security system. Each directory lists the configuration files used by the default security system.

```
JEUS_HOME/domains/<domain name>
|--config
  |--security
    |--security.key
    |--policy
    |--security-domains.xml
    |--SYSTEM_DOMAIN
      |--accounts.xml
      |--policies.xml
```

security

The following describes each directory and file.

Classification	Description
security.key	The file that stores the key for synchronous encryption algorithm. The file will be created when the encryption is executed for the first time.
policy	The Java Permission configuration file. It is used in Java SE Security Manager, separate from the JEUS security system.
security-domains.xml	The file that stores the configuration for JEUS security domains.
SYSTEM_DOMAIN	<p>A domain used by the JEUS server to check for user authentication and authorization. This domain contains Permissions to start and terminate JEUS, and JEUS system administrator accounts.</p> <ul style="list-style-type: none">◦ accounts.xml: Stores user information.◦ policies.xml: Stores security policy information (permission grating data).



To apply different security policies to applications, a separate security domain directory must be created and configured.

2.2. Configuring the Security System Domain

Domains can be configured by manually modifying the domain.xml or security-domains.xml file. Other configurations than the user account and security policy settings cannot be changed dynamically. Therefore, when a security domain is added or security service configurations are changed, the server must be restarted to apply the changes.



All configurations related to security apply to all servers in the JEUS domain. All servers must be restarted in order to apply the security configurations, except for user account and security policy configurations.

This section describes how to define a security domain by editing the xml files.

2.2.1. Configuring XML

To directly modify the XML file, use the following tags from JEUS_HOME/domains/<domain name>/config/domain.xml and JEUS_HOME/domains/<domain name>/config/security/security-domains.xml files.

XML configuration methods for a domain are defined in jeus-domain.xsd, jeus-security.xsd, and security-domains.xsd in the JEUS_HOME/lib/schemas/jeus/supportLocale/ko directory. The parent tag

is **<security-manager>**, and it has the following child tags for domain configurations. Zero or more child tags can be used.

Configuring a Security System Domain: <domain.xml>

```
<security-manager>
  <default-application-domain>SYSTEM_DOMAIN</default-application-domain>
  <security-domain-names>
    <security-domain-name>SYSTEM_DOMAIN</security-domain-name>
    ...
  </security-domain-names>
</security-manager>
```

The following describes configuration tags.

- <security-manager>

Sets the security domains that will be registered with JEUS. For more examples of <security-manager>, refer to [Configuring Security Domain Components](#).

Tag	Description
<connect-retries>	Sets the connection retry count in JEUS Security NetworkService. (Default value: 10)
<password-validator>	When specifying a password for a JEUS account, password validity check is configured to enhance password security.
<default-application-domain>	Sets the default domain name that will be used in Jakarta EE applications. (Default value: SYSTEM_DOMAIN)

- <security-domain-names>

Specifies the name of security domains to be used in JEUS.

Tag	Description
<security-domain-name>	Sets the security domain name. It is a value referencing the 'name' field in the security domain configuration.

Configuring a Security System Domain: <security-domains.xml>

```
<security-domains>
  <security-domain>
    <name>SYSTEM_DOMAIN</name>
    <authentication />
    <authorization />
    ...
  </security-domain>
</security-domains>
```

The following describes configuration tags.

- <security-domain>

Configure the security domains that will be registered with JEUS. For more examples of <security-domain>, refer to [Configuring Security Domain Components](#).

Tag	Description
<name>	Sets the security domain name.
<authentication>	Defines the authentication service.
<authorization>	Defines the authorization service.
<identity-assertion>	Defines the identity assertion.
<credential-mapping>	Defines the credential mapping service.
<credential-verification>	Defines the credential verification service.
<audit>	Defines the audit service.
<subject-validation>	Defines the subject validation service.

2.2.2. Configuring User Accounts and Security Policies

This section describes how to configure user accounts and security policies using the default XML file.

It is possible to store the user account and security policy information in the database as well as in the XML file. For more information, refer to [Using Database](#) for user accounts and [Using Database](#) for security policies.

The following are steps to configure accounts and policies.

1. To configure a new security domain, a new directory must be created under the JEUS_HOME/domains/<domain name>/config/security directory. The name of the new directory should match the name of the domain that you wish to create. By convention, domain names use all capital letters and are appended with the string "_DOMAIN". For example, the name "DEFAULT_APPLICATION_DOMAIN" can be used as a domain name.

To create a domain with the name DEFAULT_APPLICATION_DOMAIN, execute the following command.

```
$ mkdir ${JEUS_HOME}/domains/domain1/config/security/DEFAULT_APPLICATION_DOMAIN
```



domain1 should be replaced with the actual JEUS domain name.

2. After creating the new domain directory, you must create a number of configuration files in the

directory.

The best way is to copy existing configuration files from the existing domain directory and modify these files according to your need. Enter the following commands in a single line. The following sections will explain how to modify these copied configuration files.

```
$ cp ${JEUS_HOME}/domains/domain1/config/security/SYSTEM_DOMAIN/*.*  
    ${JEUS_HOME}/domains/domain1/config/security/DEFAULT_APPLICATION_DOMAIN
```

By default, SYSTEM_DOMAIN already exists under the JEUS_HOME/domains/<domain name>/security directory. SYSTEM_DOMAIN is the domain used by the JEUS server to perform authentication and authorization. Among other things, this domain contains the JEUS system "administrator" account and permissions for starting and terminating the JEUS server.

3. Add the new security domain name to JEUS_HOME/domains/<domain name>/config/domain.xml and JEUS_HOME/domains/<domain name>/config/security/security-domains.xml, respectively. In the case of domain.xml, it must be added to <security-manager><security-domain-names>, and in the case of security-domains.xml, it must be added to <security-domains><security-domain>.
4. The newly created domain will be applied when JEUS is restarted.

The security domain, SYSTEM_DOMAIN, will always be included regardless of the configurations in the domain.xml file. In general, you can create a directory with the same name as the domain under the JEUS_HOME/domains/<domain name>/config/security directory. In this directory, you can define the security policy information (policies.xsd) and account information (accounts.xsd) of the Repository, which needs to be defined for each domain. You can also use the domain.xml file to register the Security Service for each domain.

2.3. Configuring Security Domain Components

This section describes how to configure security domain components, excluding security services.

2.3.1. Configuring XML

The security services that are loaded to each security domain are defined in **domain.xml** and **security-domains.xml**.

The security configurations are defined by an XML schema in jeus-security.xsd and security-domains.xsd which are under the JEUS_HOME/lib/schemas/jeus/supportLocale/ko directory.

Inside the <security-domains> tag, add <security-domain> child tags related to security settings. One or more <security-domain> tags can be used, and each tag defines a security domain used in JEUS.

Configuring a Security System Domain: <domain.xml>

```
<?xml version="1.0" encoding="UTF-8"?>  
<domain version="9.0" xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
```

```

. . .
<security-manager>
  <default-application-domain>DEFAULT_APPLICATION_DOMAIN</default-application-domain>
  <security-domain-names>
    <security-domain-name>SYSTEM_DOMAIN</security-domain-name>
    <security-domain-name>DEFAULT_APPLICATION_DOMAIN</security-domain-name>
    . . .
  </security-domain-names>
</security-manager>
. . .
</domain>

```

Security System Service Configurations: <security-domains.xml>

```

<?xml version="1.0" encoding="UTF-8"?>
<security-domains xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9.0">
  <security-domain>
    <name>SYSTEM_DOMAIN</name>
    . . .
  </security-domain>
  <security-domain>
    <name>DEFAULT_APPLICATION_DOMAIN</name>
  </security-domain>
</security-domains>

```



Inside the <security-domain> tag, if you only configure the <name> tag and do not configure other Service Provider information, the default security services will be used for the corresponding domain.

Descriptions of child tags of <security-domain> are as follows.

- <name> (required)

The security domain name.

- <cache-config> (0 or more, optional)

Defines the cache policy value that is to be applied to the security repository service in the domain.

Tag	Description
<min>	The minimum cache entry size that will be applied to the repository service.
<max>	The maximum cache entry size that will be applied to the repository service.
<timeout>	The cache entry timeout value that will be applied to the repository service.

- <keystore-config>(0 or more, optional)

Defines the keystore file information that will be applied to the security service in the domain.

If child configuration values do not exist and '-Djeus.ssl.*' or '-Djavax.net.ssl.*' is not configured, the default value will be applied.

Tag	Description
<keystore-path>	The path to the keystore file to be applied to the current domain. (JEUS_HOME/domains/<domain name>/config/security/keystore)
<keystore-alias>	The keystore alias. When there are multiple keyEntry certificates in the keystore file, the keystore entries are accessed via unique aliases.
<keystore-password>	The keystore file password for the current domain. (Default value: changeit).
<keystore-keypassword>	The password for the keystore file for the current domain. (Default value: Same with the value of <keystore-password>)
<truststore-path>	The path to the truststore file. (JEUS_HOME/domains/<domain name>/config/security/truststore)
<truststore-password>	The truststore file password for the current domain (Default value: changeit).

2.4. Configuring Security Services

JEUS security system supports plugin type of authentication and authorization services. This section describes how to configure security domain components, including security services, by modifying the XML file.

2.4.1. Configuring XML

The security services that will be loaded for each security domain are set in the **security-domains.xml** file as follows.

The following is an example of a security configuration file.

Security System Service Configurations: <security-domains.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<security-domains xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9.0">
  <security-domain>
    <name>SYSTEM_DOMAIN</name>
    <authentication>
      <repository-service>
        <xml-file-repository>
          <config-file>
            <filename>accounts.xml</filename>
            <filepath>
              ${JEUS_HOME}/domains/domain1/config/security/
            </filepath>
          </config-file>
        </xml-file-repository>
      </repository-service>
    </authentication>
  </security-domain>
</security-domains>
```

```

        </config-file>
    </xml-file-repository>
</repository-service>
</authentication>
</security-domain>
<security-domain>
    <name>DEFAULT_APPLICATION_DOMAIN</name>
</security-domain>
</security-domains>

```

Descriptions of child tags of <security-domain> are as follows.

- <name> (required)

The security domain name.

- <authentication> (0 or more, optional)

Defines the Credential Verification service that applies to the domain.

- <repository-service>

Defines the services according to the user information storage type for authentication.

Tag	Description
<xml-file-repository>	When the user information is saved in XML file.
<database-repository>	When the user information is saved in database.
<custom-repository>	When the user information loading method is applied by implementing the extended repository service that inherits the jeus.security.spi.AuthenticationRepositoryService SPI.

- <jaas-login-config>

Registers the JAAS service that applies to the domain.

- <callback-handler-class> : JAAS Callback Handler Factory class name.
- <login-module> : Configuration related to the LoginModule.

Tag	Description
<login-module-classname>	Class name that contains the package that implements the LoginModule.

Tag	Description
<control-flag>	Property definition for controlling the authentication stack by defining one of the following four properties: <ul style="list-style-type: none"> ◦ required ◦ requisite ◦ sufficient ◦ optional
<option> (0 or more)	Definitions of property values that will be applied when initializing the LoginModule.

- <custom-authentication-service>

Defined when applying the default authentication service that is extended by inheriting the `jeus.security.spi.AuthenticationService` SPI. Follow the [Custom Security Services](#) configuration methods.

- <authorization> (0 or more, optional)

Defines the authorization service that applies to the domain.

- <repository-service>

Defines service according to the policy information repository type used for authorization.

Tag	Description
<xml-file-repository>	When the policy information is saved in XML file.
<database-repository>	When the policy information is saved in a database.
<custom-repository>	When the policy information loading method is applied by implementing the extended repository service that inherits the <code>jeus.security.spi.AuthorizationRepositoryService</code> SPI.

- <custom-authorization-service>

Defined when applying the default authorization service that is extended by inheriting the `jeus.security.spi.AuthorizationService` SPI. Follow the [Custom Security Services](#) configuration methods.

- <jacc-service>

Defined when applying the JACC 2.0-based authorization service.

- <identity-assertion>(0 or more, optional)

Defines the IdentityAssertion service that applies to the domain.

- <default-identity-assertion-service>

Defines the basic Identity Assertion Service that is provided by the JEUS Security Framework.

- <x509-identity-assertion>: Supports Identity Assertion Service for the X509Certificate token.

Tag	Description
<config-file>	Specifies the location of the file that defines the information needed to perform user mapping of the X509Certificate token. (Default value: user-cert-map.xml under the DOMAIN directory)
<default-user-mapper>	Defines a delimiter for the attribute value (attribute-value-delimiter), attribute Type(attribute-type), or attribute type(cert-attr-type) value for the X509Certificate Token value.

- <custom-identity-assertion-service>

Defined by implementing the Identity Assertion Service that inherits jeus.security.spi.IdentityAssertionService SPI. Follow the [Custom Security Services](#) configuration methods.

- <credential-mapping> (0 or more, optional)

Defines the CredentialMapping service that applies to the domain.

- <default-credential-mapping-service>

Supports the basic Credential Mapping Service that is provided by the JEUS Security Framework.

- <x509-credential-mapping> : Supports the Credential Mapping Service for the x509Certificate.

Tag	Description
<truststore-path>	Defines the path for truststore file that is to be applied to the current domain.
<truststore-password>	Defines the password for truststore file that is to be applied to the current domain.

- <custom-credential-mapping-service>

Defined by implementing the Credential Mapping Service that inherits jeus.security.spi.CredentialMappingService SPI. Follow the [Custom Security Services](#) configuration methods.

- <credential-verification> (0 or more, optional)

Defines the Credential Verification service that applies to the domain.

- <default-credential-verification-service>

Supports the basic Credential Verification Service that is provided by the JEUS Security Framework.

Tag	Description
<password-verification>	Defines the verification service for the PasswordFactory class.
<jeus-certificate-verification>	Defines the verification service for the X509Certificate.

- <custom-credential-verification-service>

Defined by implementing the Credential Verification Service that inherits jeus.security.spi.CredentialVerificationService SPI. Follow the [Custom Security Services](#) configuration methods.

- <audit> (0 or more, optional)

Defines the EventHandlingService that is to be applied to the corresponding domain. This is used to collect information about events that occurred in JEUS Security Framework.

Tag	Description
<default-audit-service>	Defines the event log level and the file path where event information is saved. <ul style="list-style-type: none"> • config-file: Log file path. • audit-level: Log level.
<custom-audit-service>	Defined by implementing the Audit Service that inherits jeus.security.spi.EventHandlingServiceService SPI. Follow Custom Security Services configuration methods.

Custom Security Services

The following describes how to configure custom security services. For more information about the security service types provided by JEUS and custom security service development, refer to [Developing Custom Security Service](#).

- <classname> (Required)

Name of the Java class that implements the Custom Security Service. This class should contain a default public no-argument constructor, and should inherit the jeus.security.spi package's SPI or directly inherit the jeus.security.base.Service class.

- <property> (0 or more)

The jeus.security.base.PropertyHolder interface (that is implemented by the

jeus.security.base.Service class) can be used to configure property as a name-value pair for security services. This property is used to initialize each of the security services.

It contains the following two child tags.

Tag	Description
<name>	Property name.
<value> (optional)	String property value for the property name.

2.5. Configuring the Security System User Information

In the default security configurations, user data is read from the **accounts.xml** file.

The following is the file path.

```
JEUS_HOME/domains/<domain name>/config/security/<security domain name>/accounts.xml
```

Here, *<domain name>* is the name of the domain and *<security domain name>* is the name of the security domain for which the users are managed.

2.5.1. Configuring XML

The XML schema of **accounts.xml** is accounts.xsd, which is in the JEUS_HOME/lib/schemas/jeus/supportLocale/ko directory.

In the accounts.xml file, there are 0 or more <user> and <group> tags inside the <accounts> tag. They represent a user and a group, respectively.

Security System User Information Configuration: <accounts.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accounts xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <users>
    <user>
      <name>user1</name>
      <password>{AES}mnG6ItxF0/WQ1E2YzIZ7sA==</password>
      <group>group1</group>
    </user>
    <user>
      <name>user2</name>
      <password>{DES}7dJ0KDTGQNpSnQAPYBNnmA==</password>
    </user>
    <user>
      <name>user3</name>
      <password>{DES}7dJ0KDTGQNpSnQAPYBNnmA==</password>
      <group>nested_group</group>
    </user>
    <user>
  
```

```

        <name>user4</name>
        <password>{SEED}d12EePMAcnxPYbIyknuZkA==</password>
        <group>nested_group</group>
    </user>
</users>
<groups>
    <group>
        <description>Group1</description>
        <name>group1</name>
        <subgroup>nested_group</subgroup>
    </group>
    <group>
        <description>For NestedGroup</description>
        <name>nested_group</name>
    </group>
</groups>
</accounts>

```

The following describes configuration tags.

- <user>

Each <user> tag has the following child tags.

Tag	Description
<description> (optional)	Description of the <user> (string).
<name> (required)	Name of the <user>. (ex : user name, user id)
<password> (0 or more, optional)	Password of the <user>. The value that is encoded by using a particular algorithm. The algorithm or encoding method applied to <password> will be specified in '{}'. Refer to Configuring Password Security .
<group> (0 or more, optional)	The name of the group that the <user> belongs to. A user can be included in multiple groups. Also, the group must designate one of the groups defined in the <group> tag, described next.

- <group>

Each <group> tag defines a group and contains the following child tags.

Tag	Description
<description> (optional)	Description of the <group> (string).
<name> (required)	Name of the <group>.
<subgroup> (0 or more, optional)	Defines the nested group names so that the corresponding group role can be applied uniformly.

2.5.2. Using Database

This section explains how to configure authentication using the database table defined by JEUS.

The following example illustrates how to use the data source defined in JEUS.

Configuring Users Using Database: <security-domains.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<security-domains xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9.0">
  . . .
  <security-domain>
    <name>MY_DOMAIN</name>
    <authentication>
      <repository-service>
        <database-repository>
          <datasource-id>auth</datasource-id>
        </database-repository>
      </repository-service>
    </authentication>
  </security-domain>
  . . .
</security-domains>
```

In the previous example, the data source ID must be configured using the **<datasource-id>** element. The data source ID is registered in the security-domains.xml file that will be used for authentication. Authentication can also be performed by directly communicating with the database through the DriverManager without using the JDBC that is provided by JEUS. To do this, change the <datasource-id> element block of the previous example as follows.

The following example illustrates how to configure the database repository when not using the JEUS JDBC.

Without Using JEUS JDBC: <security-domains.xml>

```
<security-domain>
  ...
  <repository-service>
    <dbdriver-config>
      <vendor>oracle</vendor>
      <driver>oracle.jdbc.OracleDriver</driver>
      <url>jdbc:oracle:thin:@127.0.0.1:1521:ORA9I</url>
      <username>scott</username>
      <password>{base64}dGlnZXI= </password>
    </dbdriver-config>
  </repository-service>
  ...
</security-domain>
```



As applications frequently establish database connections and close them in this way, there may be some drop in performance. So, it is recommended to use JEUS

property	value
min length	4
max length	10
include special characters	false
include digit characters	false
include capital characters	false
include small characters	false
exclude user id	false

The following describes each configuration item when the 'Validator Type' is set to 'Default Password Validator':

Item	Description
min	Minimum length of a password. (range: 1 ~ 255, default value: 1)
max	Maximum length of a password. (range: 1 ~ 255, default value: 255)
special	Indicates that at least one special character must be included.
digit	Indicates that at least one numeric value must be included.
capital	Indicates that at least one uppercase letter must be included.
small	Indicates that at least one lowercase letter must be included.
excludeId	Indicates that the user ID should not be included.

Once the items are configured, the configurations are saved in **domain.xml** as in the following:

Configurations for Default Password Validator: <domain.xml>

```
<domain>
  . . .
  <security-manager>
    . . .
    <password-validator>
      <default-password-validator>
        <minLength>4</minLength>
        <maxLength>10</maxLength>
        <force-special-character>true</force-special-character>
        <force-digit>true</force-digit>
        <force-capital-letter>true</force-capital-letter>
        <force-small-letter>true</force-small-letter>
        <deny-username>true</deny-username>
      </default-password-validator>
    </password-validator>
    . . .
  </security-manager>
  . . .
</domain>
```

After all the configurations are saved, they are dynamically applied as password rules. These rules must be met when a user creates a new password or updates an existing one.

• Configuring Custom Password Validator

A user can create a Custom Password Validator by implementing the `jeus.util.PasswordValidator` interface. The user must directly implement this interface to create a class with the desired functionality, package it into a JAR file, and place it in the 'DOMAIN_HOME/lib/application' directory.

The `jeus.util.PasswordValidator` interface is as follows:

```
public interface PasswordValidator {
    boolean validatePassword(String id, String password);
}
```

For a class that implements the `PasswordValidator` interface, the `validatePassword(String id, String password)` method takes as parameters the ID and password of the user who creates an account or changes the password. The method then checks the validity of the password and returns either true or false depending on the validation result.

Once the JAR file containing the class is placed in the `DOMAIN_HOME/lib/application` directory, use `jeusadmin` commands to configure a Custom Password Validator.

This item can be configured using the `jeusadmin` commands related to Custom Password Validator, which are **add-custom-password-validator**, **remove-custom-password-validator**, and **show-custom-password-validator**. For information about the commands, refer to "Part II. Console Commands and Tools" in JEUS Reference Guide.

```
[MASTER]domain1.adminServer>add-custom-password-validator -class MyValidator
Custom password validator [MyValidator] is added successfully.
Check the results using "show-custom-password-validator".
[MASTER]domain1.adminServer>show-custom-password-validator
=====
+-----+
|               custom password validator class names               |
+-----+
| MyValidator |
+-----+
=====
```

Once the item is configured, the configuration is saved in **domain.xml** as in the following:

Configurations for Custom Password Validator: `<domain.xml>`

```
<domain>
  . . .
  <security-manager>
    . . .
    <password-validator>
      <custom-password-validator>
        <class-name>MyValidator</class-name>
        <class-name>MyValidator2</class-name>
      </custom-password-validator>
    </password-validator>
```

```
    . . .  
    </security-manager>  
    . . .  
</domain>
```

The server must be restarted for the Custom Password Validator configuration to take effect. After the server restart, the password rules must be met when a user creates a new password or updates an existing one. If more than one class has been defined to Custom Password Validator, all password rules contained in all the classes must be met for password validation.



The server will not restart if there is no JAR file of the user-defined class in the DOMAIN_HOME/lib/application directory. In this case, the user must set the relevant items in **domain.xml** correctly for the server to restart.

Encryption Algorithms

When configuring a user, the user password must be set. The password can be stored in plain-text, but for security reasons, it is better to encrypt the password using an encryption algorithm.



It is recommended to use a one-way encryption algorithm in accordance with the national encryption guidelines of Korea.

Passwords can be encrypted by using the **set-password** command in jeusadmin, or directly by using JEUS_HOME/bin/encryption.

To manually encrypt the password, use the following format.

```
{algorithm}encrypted password
```

The key value of the symmetric-key algorithm is stored in JEUS_HOME/domains/<domain name>/config/security/security.key.

The following are descriptions of algorithms that can be used for password encryption.

Item	Description
AES/DES/DESeed/SEED/Blowfish	cryptographic symmetric-key algorithm.
base64	Encoding algorithm. Since the information encrypted in Base64 can be easily decoded by any one, it is not secure.
SHA	Hash algorithm. This one-way encryption algorithm cannot be decrypted.

Item	Description
SSHA	Salted Hash Algorithm. This one-way encryption algorithm cannot be decrypted. It enhances the conventional hash algorithm by adding salting, which significantly increases resistance to reversal through rainbow tables.

The user can specify the key size for an encryption algorithm. The key size is managed as a global system property by an administrator.

The default key size is 256 bits and can be changed using the `jeus.security.keylength` system property. For example, the `"-Djeus.security.keylength=256"` option allows an encryption algorithm with a 256-bit key.

If the key size set with the system property is larger than the maximum size supported by an encryption algorithm, the supported maximum size will be used. For example, since AES supports only 128-bit, 192-bit, and 256-bit keys, AES512 will become AES256.

If a specified key size is smaller than the maximum size supported by an encryption algorithm, but is not a valid size, `EncryptionException` occurs. For example, since AES supports only 128-bit, 192-bit, and 256-bit keys, the AES200 setting causes `EncryptionException`.



After the system property is modified, passwords must be initialized.

SecretKey File Management Using the Master Password

When encrypting a password using the encryption tool (located in `JEUS_HOME/bin`), the `SecretKey` information that is applied to an encryption file can be stored in `security.key` file grouped by algorithm types. A master password can be used to encrypt the `security.key` file.

The `security.key` file is placed in the following path. When configuring a domain environment, the `security.key` file must be moved to another node.

```
JEUS_HOME/domains/<domain name>/config/security/security.key
```

When you encrypt the DB password, which needs to be registered in JEUS, a client needs a key to decrypt it. At this point, you can configure the secret key file path by using the system property.

When the master password is specified in the secret key file, this master password can also be configured using the system property. The property name for the key path is `jeus.security.keypath`. To specify the master password, use the `jeus.security.master` property. These properties can also be used when starting JEUS. However, for security reasons, it is recommended to use the prompt (standard input) to enter the master password.

2.5.4. Cached Login Information

The user information must be entered when starting a server using the JEUS script or by accessing the server using jeusadmin. In such case, the user information can be cached so that the user information does not need to be entered every time.



The information is stored encrypted in AES in USER_HOME/.jeusadmin/.jeuspasswd. The JEUS_HOME/bin/security.key file is used for encoding and decoding. Since the AES encoding is used to store sensitive user information, ID/password, it is not recommended for use.

Only the login information, that was authenticated, is stored in the cache.

When executing the **connect** command in jeusadmin or JEUS script at server startup, if the option **-cachellogin** is added with the user information, the login information will be stored in the file. When JEUS script is used, the login information is stored using the key, **<domain name>:<server name>**.

When JEUS script is executed, if login information with the same domain and server name exists, the user information will be automatically filled without having to re-enter the user information. If the user manually enters the information, even if there is already cached login information, the cached information will be ignored.

The following example shows the stored login information.

```
#Warning: We don't recommend to use this on Production Environment.  
domain1:user1  
gEPaqBz6BaAxWxdSXf8wZNPLsWkysgcov/KJnHvDeduKRvTAOb7F6zRaPHc2zLBUIUi46FQFWn14mQiEIUbG9UEe4yZrsRri7yS9q  
i+7EwA=
```

The stored login information can be deleted using the **remove-login-cache** command, an off-line jeusadmin command.

2.6. Configuring Security System Policies

When configuring default securities, the policy data (authorization data) is read from the file, **policies.xml**.

The file is in the following path.

```
JEUS_HOME/domains/<domain name>/config/security/<security domain name>/policies.xml
```

<domain name> is the domain name, and **<security domain name>** is the domain name where the policy will be applied.

2.6.1. Configuring XML

The XML schema of the policies.xml file is **policies.xsd**, and it is in the JEUS_HOME/lib/schemas/jeus/supportLocale/ko directory. policies.xml can contain 0 or more **<policy>** tags. Each **<policy>** tag represents an individual policy (authorization data).

Security System Policy Configuration: <policies.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<policies xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <policy>
    <role-permissions>
      <role-permission>
        <principal>user1</principal>
        <role>administrator</role>
      </role-permission>
      <role-permission>
        <principal>user1</principal>
        <role>teller</role>
        <actions>9:00-17:00</actions>
        <classname>
          jeus.security.resource.TimeConstrainedRolePermission
        </classname>
      </role-permission>
    </role-permissions>
    <resource-permissions>
      <context-id>default</context-id>
      <resource-permission>
        <role>teller</role>
        <resource>bankdb</resource>
        <actions>select, update</actions>
      </resource-permission>
      <resource-permission>
        <role>administrator</role>
        <resource>jndi</resource>
        <actions>modify, delete</actions>
      </resource-permission>
      <resource-permission>
        <resource>jndi</resource>
        <actions>lookup</actions>
        <unchecked/>
      </resource-permission>
      <resource-permission>
        <role>administrator</role>
        <resource>jeus.*</resource>
        <actions>*</actions>
      </resource-permission>
    </resource-permissions>
  </policy>
</policies>
```

The following describes configuration tags.

- **<role-permissions>** (0 or 1)

Provides information about principal-to-role mappings. Multiple **<role-permission>** tags are

nested inside this tag.

The <role-permission> tag consists of the following child tags.

Tag	Description
<principal> (0 or more)	Principal name that owns the current role-permission.
<role> (1ea, required)	Role name.
<actions> (optional)	Actions for the Role.
<classname> (optional)	The Java class name that implements java.security.Permission, which will be used for role permission. If omitted, by default, jeus.security.resource.RolePermission is used. For more information about configuring Custom Permission, refer to "Custom Permission Implementation and Configurations" .
<excluded> (optional)	Empty tag (<xxx/>)with no value. If specified, the role permission is excluded (this means that the role implied by the permission will be granted to no one).
<unchecked> (optional)	Empty tag. If specified, the role permission is unchecked (this means that the role implied by the permission will be granted to everyone).

- <resource-permissions> (0 or more)

These tags contain information about role-to-resource mapping as follows:

- <context-id> (optional)

Context ID (string). Context is the scope for authorization checks. The default Context ID for JEUS resources used by JEUS system is "default".

- <resource-permission> (0 or more)

The <resource-permission> tag consists of the following tags.

Tag	Description
<role> (0 or more)	Role name that owns the current resource-permission.
<resource> (1ea, required)	Resource name.
<actions> (optional)	Actions for the resource.

Tag	Description
<classname> (optional)	The Java class name that implements java.security.Permission which will be used for Resource Permission. If omitted, jeus.security.resource.ResourcePermission is used by default. For more information about configuring Custom Permission, refer to "Custom Permission Implementation and Configurations" .
<excluded> (optional)	Empty tag with no value. If specified, the resource permission is excluded. (this means that the resource implied by the permission will be granted to no one).
<unchecked> (optional)	Empty tag with no value. If specified, the resource permission is unchecked. (this means that the resource implied by the permission will be granted to everyone).

Normally the policies.xml file is exclusively used to configure permissions for JEUS Server Resource (like JNDI, JMS, Security Server, etc.), and not for Jakarta EE applications and modules. To configure Jakarta EE application and module permissions, you should use various DD (deployment descriptor) files. For more information, refer to [Configuring Security in Applications and Modules](#).

Custom Permission Implementation and Configurations

Whenever a permission (role permissions or resource permissions) is added to the policies.xml file, the Java class name of the permission must be specified. This is the name of a Java class that extends the java.security.Permission abstract class. Create a customized implementation subclass that extends the java.security.Permission Permission class, and specify the implementation class name in the <classname> tag of policies.xml.

This sub-section describes how to develop a new customized role permission that meets the following requirements. The new role permission should imply another role permission only if the following two conditions are met.

- The other role has the same role name as this permission.
- The current time is between the specified time limits (example: 9 am and 5 pm).

The following shows an example of a banking application where a role permission grants the principal user2 the teller role only during the 9 to 5 business hours.

1. Create the Custom Permission. Then, compile the class with javac and set the class path within the JEUS server class path. Configure the Permission in the policies.xml file. The following implements the Custom Permission class that meets the aforementioned requirements. Code details have been omitted.

Custom Permission Class: <TimeConstrainedRolePermission.java>

```
package jeus.security.resource;

import java.security.Permission;
import java.util.Calendar;
import java.util.StringTokenizer;
```

```

/**
 * A time-constrained Role permission.
 * <p>
 * With this permission implementation you can express
 * things such as "X can only be in the role Y between
 * 09:00 AM to 05:00 PM".
 */
public class TimeConstrainedRolePermission extends RolePermission {
    private String timeConstraint = "*";
    private int startTime = Integer.MIN_VALUE;
    private int endTime = Integer.MAX_VALUE;

    public TimeConstrainedRolePermission(String roleName) {
        this(roleName, "*");
    }

    public TimeConstrainedRolePermission(String roleName, String timeConstraint)
    {
        super(roleName);
        if (timeConstraint != null) {
            this.timeConstraint = timeConstraint;
            parseTimeConstraint();
        }
    }

    private void parseTimeConstraint() {
        . . .
    }

    public boolean equals(Object anotherObject) {
        . . .
    }

    public int hashCode() {
        . . .
    }

    public String getActions() {
        return timeConstraint;
    }

    public boolean implies(Permission anotherPermission) {
        if (this.timeConstraint.equals("*")) {
            return super.implies(anotherPermission);
        } else {
            Calendar cal = Calendar.getInstance();
            int curHour = cal.get(Calendar.HOUR_OF_DAY);
            int curMinute = cal.get(Calendar.MINUTE);
            int now = curHour * 60 + curMinute;
            if (now >= this.startTime && now <= this.endTime) {
                return super.implies(anotherPermission);
            } else {
                return false;
            }
        }
    }
}

```


The previous class inherits the `jeus.security.resource.RolePermission` class, which in turn inherits the `java.security.Permission` class. This structure promotes code reusability.

It is also possible to create another `java.security.Permission` class by inheriting the `TimeConstrainedRolePermission` class.

In the previous example, there are two types of constructors.

- The first has one parameter (role name).
- The second has two parameters (role name, time constraint)

In `java.security.Permission` class, the first parameter is name, and the second parameter is actions. Some permission implementation classes omit the second constructor that receives the actions parameter.

- The actions parameter contains the valid time period for the permission, and it has the value "09:00-17:00".
- The name parameter represents the role name, such as administrator or teller.

The heart of the implementation is in the "`implies(Permission anotherPermission)`" method, which first checks whether the current system time is within the given time constraints. And, if so, it delegates the call to the super class by calling the `super.implies()` method, and if not, it returns "false". All `implies(..)` method should return a boolean value, and whether the relevant `Permission` implies the `Permission` object passed to it.



1. For more information about the `implies()` method and the abstract class `java.security.Permission`, refer to Java SE Javadoc documents.
2. For more information about `jeus.security.resource.RolePermission`, `jeus.security.resource.TimeConstrainedRolePermission`, and `jeus.security.resource.ResourcePermission` classes, refer to [References](#) and `jeus.security.resource` package in JEUS Security Javadoc.

2. Configure the policies.xml file to use permissions.

Custom Permission Class: `<policies.xml>`

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<policies xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <policy>
    <role-permissions>
      <role-permission>
        <principal>user2</principal>
        <role>administrator</role>
        <actions>9:00-17:00</actions>
        <classname>
          jeus.security.resource.TimeConstrainedRolePermission
        </classname>
      </role-permission>
    </role-permissions>
    <resource-permissions>
```

```

        <context-id>default</context-id>
        <resource-permission>
            <role>administrator</role>
            <resource>jeus.*</resource>
            <actions>*</actions>
        </resource-permission>
    </resource-permissions>
</policy>
. . .
</policies>

```

The user, user2, is in the administrator role during the business hours of 9 am to 5 pm. The administrator role has the right to perform any actions (“*”) on all JEUS resources (“jeus.*”). Therefore, user2 can perform any operations on all JEUS resources during the business hours.

The previous rule also applies when configuring JEUS DD file in Jakarta EE application and module. Naturally, the rule also applies to role-to-resource permission and to principal-to-role permission as shown in the previous example. For more information about using customized permissions in JEUS DD, refer to [Configuring Security in Applications and Modules](#).

2.6.2. Using Database

In order to configure the policy using a database, domain service must be configured in the **security-domains.xml** file as in the following. In order to directly use the driver manager, add the <dbdriver-config> tag, instead of the <datasource-id> tag, as for the authentication setting. However, it is recommended to use JEUS JDBC data source for performance benefits.

Configuring Policies Using Database: <security-domains.xml>

```

<?xml version="1.0" encoding="UTF-8"?>
<security-domains xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9.0">
    ...
    <security-domain>
        <name>MY_DOMAIN</name>
        <authorization>
            <repository-service>
                <database-repository>
                    <datasource-id>auth</datasource-id>
                </database-repository>
            </repository-service>
        </authorization>
        . . .
    </security-domain>
    . . .
</security-domains>

```

A URL, username, and password to access the JDBC driver, which accesses the database, are required to use the AuthenticationRepositoryService that uses a database.

In the following example, a password that is encoded in Base 64 must be entered to access the Oracle DB. When entering a password applied with a certain encryption algorithm or encoding

method, use the same method as for user password of accounts.xml.

The tables that are used in the database are organized as follows:

jeus_role_principal		
PK	domain	VARCHAR(30)
PK	rolepermissionid	VARCHAR(30)
PK	username	VARCHAR(30)

jeus_role_permissions		
PK	domain	VARCHAR(30)
PK	rolepermissionid	VARCHAR(30)
	description	VARCHAR(100)

jeus_role_permission		
PK	rolepermissionid	VARCHAR(30)
PK	role	VARCHAR(30)
	actions	VARCHAR(100)
	classname	VARCHAR(100)
	excluded	VARCHAR(5)
	unchecked	VARCHAR(5)

jeus_seqno		
PK	tablename	VARCHAR(30)
PK	currentseqno	VARCHAR(30)

jeus_resource_permissions		
PK	domain	VARCHAR(30)
PK	contextid	VARCHAR(30)
PK	resourcepermissionid	VARCHAR(30)
	description	VARCHAR(100)

jeus_resource_permission		
PK	domain	VARCHAR(30)
PK	res	VARCHAR(30)
	actions	VARCHAR(100)
	classname	VARCHAR(100)
	excluded	VARCHAR(5)
	unchecked	VARCHAR(5)

jeus_resource_role		
PK	domain	VARCHAR(30)
PK	rolepermissionid	VARCHAR(30)
PK	resourcepermissionid	VARCHAR(30)

Database Table Structure to Save Policies

If the tables do not exist in the database, create a DB table by using the JEUS_HOME/templates/security/dbrealm.sql.template file. The basic policy is granted the administrator role in "SYSTEM_DOMAIN", thereby possessing Resource authority for "jeus.*". The default principal included in the administrator role is jeus as mentioned in [Using Database](#).

2.7. Configuring Additional Settings

This section describes how to add additional configurations other than subject and policy.

2.7.1. Configuring Java SE SecurityManager

The use of a JavaSE SecurityManager in JEUS can increase the platform reliability, but might undermine the performance. Usually, all core JEUS codes as well as the Jakarta EE application and module code deployed to JEUS may be treated as completely “trusted” code, and thereby avoid the overhead incurred by the SecurityManager. This mode of operation, with no SecurityManager, is the default mode in JEUS.

However, there may be cases where the increased reliability and code-level security provided by a JavaSE SecurityManager is considered more important than the performance enhancement.

For example, the administrator may have to deploy some untested or otherwise suspicious Jakarta EE applications or modules to JEUS. In such case, it may be desirable to protect your machines and environment by boosting the code level protection at the expense of performance by using JavaSE SecurityManager.

In order to use a JavaSE SecurityManager with JEUS, define jvm-option in the domain.xml file for a particular server as follows.

```
-Djava.security.manager  
-Djava.security.policy=${JEUS_HOME}/domains/domain1/config/security/policy(for UNIX)
```

The policy file is JEUS_HOME/domains/<domain name>/config/security/policy and the its content is as follows:

Java SE SecurityManager Configurations: <policy>

```
grant codeBase "file:${jeus.home}/lib/system/*" {  
    permission java.security.AllPermission;  
};  
  
grant {  
    permission java.net.SocketPermission "127.0.0.1:1024-",  
        "connect, accept, connect, listen, resolve";  
    permission java.security.SecurityPermission "runTrustedLogin", "read";  
    permission java.security.SecurityPermission "loginCodeSubject", "read";  
    permission java.security.SecurityPermission "putProviderProperty.*";  
    permission java.security.SecurityPermission "insertProvider.*";  
    permission javax.management.MBeanPermission "*", "*";  
};
```

The JavaSE SecurityManager is completely independent from JEUS security system. The JEUS security system does not provide protection at the code level (permission configuration that can execute code), but instead it provides security at the user level (the identity of the process that is running and what it is allowed to do).

The only overlap between the two is that, under special conditions, the JEUS security system can check with the JavaSE SecurityManager for some code level permission (see above) in order to protect JEUS from malicious or bug-filled Servlet and EJB code.

2.7.2. Configuring JACC Provider

For details of configuring the JACC 2.0, refer to [Using JACC Provider](#).

2.7.3. Configuring information to Grant Identity

If IdentityAssertionService is supported, it is read from the **cert-user-map.xml** file that contains the mapping information between a certificate and user. The file is in the following location.

```
JEUS_HOME/domains/<domain name>/config/security/<security domain name>/
```

Here, *<domain name>* is the name of the domain and *<security domain name>* is the name of the security domain for which the users are managed.

The XML schema of user-cert-map.xml is user-cert-map.xsd, and its path is JEUS_HOME/lib/schmas/jeus/supportLocale/ko.

In the file, the parent tag, **<cert-user>**, can have **<user>**, **<cert>**, and multiple **<cert-user>** child tags. Each contains the property information for user to certificate mapping.

Configuring Information to Grant Identity: <cert-user-map.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<cert-user-map xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <cert-user>
    <username>user1</username>
    <cert>
      <subjectDN>user1DN</subjectDN>
    </cert>
  </cert-user>
</cert-user-map>
```

Each <cert-user> tag has the following child tags.

- <username> (required)

The user name that is mapped to an attribute value of the certificate. (Ex: User name, User ID)

- <cert>

Each <cert> tag has the following child tags, which define the certificate mapping information for the user. It defines the value that maps to user name using child attribute value that ensure a unique ID for granting the identity of the certificate, which is included in the truststore file.

Tag	Description
<alias> (optional)	Defines the alias for certificate within the keystore. (string)
<subjectDN> (optional)	Defines the subejctDN for certificate the within keystore. (string)

Tag	Description
<SKI> (optional)	Defines the SKI for certificate within the keystore. (string)
<issuer> (optional)	Defines the certificate issuer within the keystore. (string)
<serialNo> (optional)	Defines the certificate serial number within the keystore. (string)

2.7.4. Configuring Identity Certificate Information

The JEUS security system provides an API, through the UserCertMappingService, that can be used to obtain certificate information of the authenticated user's identity. If UserCertMappingService class is supported, the user data is read from the **user-cert-map.xml** file, where additional mapping information exists for the certificate grouped by users. The file is in the following path.

```
JEUS_HOME/domains/<domain name>/config/security/<security domain name>/
```

Here, *<domain name>* is the name of the domain and *<security domain name>* is the name of the security domain for which the users are managed.

The XML schema of user-cert-map.xml is user-cert-map.xsd, and its path is JEUS_HOME/lib/schmas/jeus/supportLocale/ko.

In the file, the parent tag, **<user-cert-map>**, can have 0 or more **<user-cert>** tags. Each contains the property information to obtain a user certificate from the Keystore file.

Configuring the Certificate Information for Identity: **<user-cert-map.xml>**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user-cert-map xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <user-cert>
    <username>user1</username>
    <alias>alias1</alias>
    <keypassword>changeit</keypassword>
    <secretkey>
      <keyname>testkeypass</keyname>
      <keyalgorithm>AES</keyalgorithm>
      <keyvalue>bjhTUjJvSXRTOGlkVEdlNHJnM2N3Vn1jSDZXV0JkYz0=</keyvalue>
    </secretkey>
  </user-cert>
</user-cert-map>
```

Each **<user-cert>** tag has the following child tags.

- **<username>** (required)

Defines the user name for the certificate within the keystore. Since this is the primary identity, it must be unique. (ex: user name, user ID)

- **<alias>** (required)

Defines the alias for certificate within the keystore.

- <keypassword> (optional)

Defines the keypassword to obtain the private key of the certificate within the keystore. (ex: changeit)

- <secretkey> (optional)

Defines the private key.

Each <secretkey> tag has the following child tags.

Tag	Description
<keyname> (required)	Defines the name of the private key. The keyname inside <user-cert-map> tag must be unique. (string)
<keyalgorithm> (required)	Indicates the key algorithm for the private key. (string)
<keyvalue> (required)	Shows the private key value in the Base64 format.

3. Configuring Security in Applications and Modules

This chapter explains how to configure security in Jakarta EE applications, EJB, and Web modules.

3.1. Overview

This section explains basic information related to security configuration for Jakarta EE applications and modules.

The basic procedure for configuring security settings of Jakarta EE application, EJB module, and web module is as follows.

1. Configure Role-to-Resource mapping for each module.
2. Configure principal-to-role mapping for each application.

3.1.1. Module Deployment vs. Application Deployment

There are two ways of deploying an application in JEUS:

- Deploy each EJB module (EJB.jar file) and Web module (.war file) independently.
- Deploy a Jakarta EE application (.ear) by compressing several EJB, Web, and connector modules (.rar files) into an EAR file.

3.1.2. Role-to-Resource Mapping

Before deploying an EJB or Web module, the assembler must first configure the role-to-resource mapping for the module. The role-to-resource mapping is also called the security constraint, which refers to mapping a resource of a module to a logical Role.

- In EJB modules, resources refer to EJB methods, and they are configured in META-INF/ejb-jar.xml
- In web modules, resources refer to the servlet URLs, and they are configured in WEB-INF/web.xml.

The following are the security constraints configured in the META-INF/ejb-jar.xml file for an EJB module:

Security Constraints Configured in EJB Module: <ejb-jar.xml>

```
<?xml version="1.0"?>
<ejb-jar xmlns="https://jakarta.ee/xml/ns/jakartaee">
  <display-name>product</display-name>
  <enterprise-beans>
    <entity>
```



```

    <ejb-name>product</ejb-name> ❶
    . . .
    <security-role-ref> ❷
        <role-name>cust</role-name>
        <role-link>customer</role-link>
    </security-role-ref>
</entity>
</enterprise-beans>
<assembly-descriptor>
    . . .
    <security-role> ❸
        <role-name>administrator</role-name>
    </security-role>
    <security-role> ❹
        <role-name>customer</role-name>
    </security-role>
    <method-permission> ❺
        <role-name>administrator</role-name>
        <method>
            <ejb-name>product</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getSecretKey</method-name>
            <method-params>
                <method-param>java.lang.Integer</method-param>
            </method-params>
        </method>
    </method-permission>
    <method-permission> ❻
        <unchecked/>
        <method>
            <ejb-name>product</ejb-name>
            <method-name>doSomeAdmin</method-name>
            <method-params>
                <method-param>java.lang.String</method-param>
            </method-params>
        </method>
    </method-permission>
    <method-permission> ❼
        <role-name>customer</role-name>
        <method>
            <ejb-name>product</ejb-name>
            <method-name>test1</method-name>
        </method>
    </method-permission>
    <exclude-list> ❽
        <method>
            <ejb-name>product</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getCustomerProfile</method-name>
            <method-params></method-params>
        </method>
    </exclude-list>
</assembly-descriptor>
</ejb-jar>

```

Descriptions of security constraints in the previous DD (deployment descriptor) file are as follows:

- ① There is an EJB entity called product.
- ② The role-to-role reference mapping is declared. The actual declared Role, customer, can be referred to by the role reference name, cust, which means that the role named cust in the EJB code corresponds to the customer Role.
- ③ The logical role administrator is declared.
- ④ The logical role customer is declared.
- ⑤ The role-to-resource mapping is declared. This enables the role administrator to call the getSecreteKey method of the remote EJB interface. According to this mapping, only Principals belonging to the administrator role can call the getSecreteKey method.
- ⑥ doSomeAdmin method is declared as unchecked. This means that anyone can call the method regardless of the Role.
- ⑦ Any principal in the role customer can call the test1 method of the product EJB.
- ⑧ The getCustomerProfile method is included in the Excluded list. This means that no role can call this method.

The configuration for Web modules is similar to that for EJB modules except that the Roles access Servlet URLs instead of EJB methods. Refer to [Configuring Web Module Security](#) for more information about security configuration for Web modules.

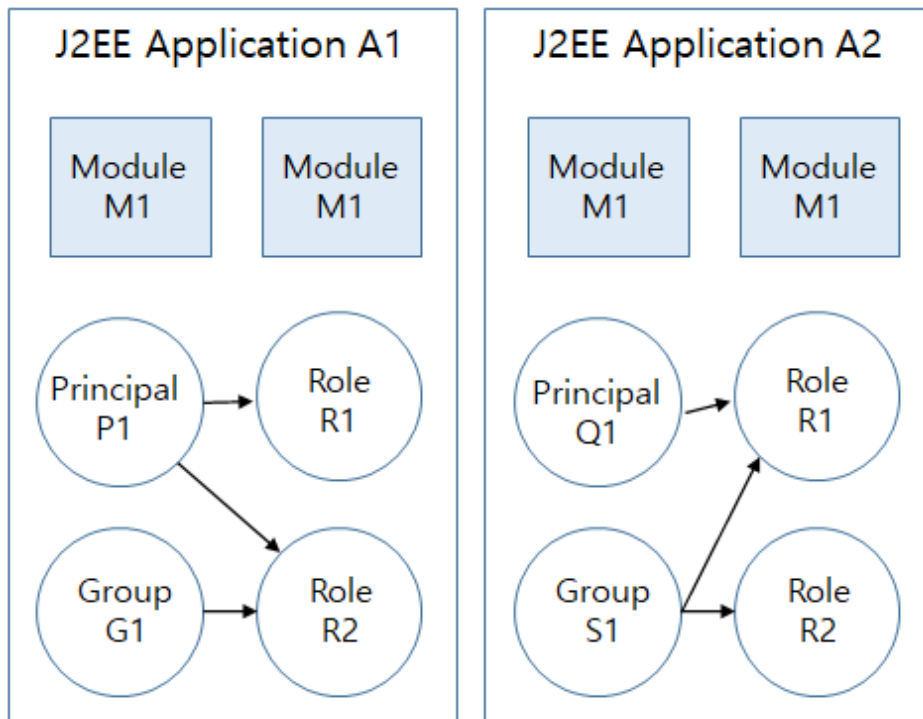
The two Roles, administrator and customer, are purely logical and do not have any meaning in the target deployment environment. Therefore, these logical Roles must be mapped to the actual users or user groups of the target system. This is referred to as principal-to-role mapping. Refer to [Principal-to-Role Mapping](#) for more information about the mapping.

3.1.3. Principal-to-Role Mapping

It is usually the task of the deployer to map the role defined in Jakarta EE modules or applications to users or user groups in the actual environment. Note that mappings from Principals to logical Roles have application scope.

For example, if two modules M1 and M2 are deployed as parts of the same application A1, the principal-to-role mappings will be shared by the two modules. Furthermore, another application A2 may have a totally different principal-to-role from A1. Even if A1 and A2 each have a role with the same name, these are recognized as different roles and are not shared.

These concepts are illustrated in the following figure.



Principal-to-Role Mapping

As shown in the previous figure, two different applications A1 and A2 have different principal-to-role mappings, and two modules M1 and M2 share the mapping included in the application.

The principal-to-role mapping is defined in the following files in JEUS.

- jeus-ejb-dd.xml, the JEUS DD file (EJB module)
- jeus-web-dd.xml (web module)
- jeus-application-dd.xml (Jakarta EE application)

The following is the example of the configuration of jeus-ejb-dd.xml for ejb-jar.xml.

Principal-to-Role Mapping: <jeus-ejb-dd.xml>

```
<?xml version="1.0"?>
<jeus-ejb-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <module-info>
    <role-permission> ①
      <principal>user1</principal>
      <role>administrator</role>
      <classname>mypackage.MyRolePermission</classname>
    </role-permission>
    <role-permission> ②
      <principal>user2</principal>
      <role>customer</role>
    </role-permission>
    <role-permission> ③
      <role>roleA</role>
      </excluded>
    </role-permission>
    <role-permission> ④
      <role>roleB</role>
      </unchecked>
    </role-permission>
  </module-info>
</jeus-ejb-dd>
```

```

    <unspecified-method-permission> ⑤
        <role>administrator</role>
    </unspecified-method-permission>
</module-info>
<beanlist>
    . . .
</beanlist>
</jeus-ejb-dd>

```

The previous example contains the following security information.

- ① The role permission that grants the principal (user) named user1 to be in the role administrator. This means that user1 is mapped to the role administrator, and consequently, user1 will inherit all the rights of the Role.

As can be seen in the previous example, custom role permission is defined in the `<classname>` tag. Refer to [Configuring Security System](#) for more information about the custom role permission.

- ② The role permission that grants the principal (user) named user2 to be in role customer. This means that user2 is mapped to the role customer and consequently, user2 will inherit all the rights of role Customer. Since no `<classname>` tag is specified, the default role permission will be used.

- ③ The role roleA is excluded from all Principals, which means that no principal can be in roleA.

- ④ The role roleB is unchecked in the authorization system, which means that all Principals are automatically included in roleB.

- ⑤ The `<unspecified-method-permission>` tag determines how unspecified EJB methods are to be handled. An unspecified EJB method is an EJB method that is not included in the `<method-permission>` tag in `ejb-jar.xml`.

The `<unspecified-method-permission>` can be handled in the following three ways: be mapped to a role (like in this example where they are mapped to the role administrator), be marked as excluded (accessible to no one) or be marked as unchecked (be accessible to everyone). The last option, unchecked, is the default value.



The configurations for Web modules (`jeus-web-dd.xml`) is similar to that for EJB modules, except that the `<unspecified-method-permission>` tag is not used for Web modules.

3.1.4. User Configurations

There are two user configuration methods:

- Use the following file for user configurations of the domain used by applications. (Refer to [Configuring the Security System User Information](#))

```
JEUS_HOME/domains/<domain name>/config/security/<security domain name>/accounts.xml
```

- Configure **accounts.xml** in the WEB-INF (Web module) or META-INF (EJB module or EAR) directories to set user configurations of EJB module (.jar file), Web module (.war file), or Jakarta EE application (.ear).



User information has application security domain scope, regardless of the way it is configured. In other words, a user in a security domain is shared by all applications using the same security domain.

3.2. Configuring EJB Module Security

This section describes in detail how to configure security for EJB modules. For information about security settings such as CSI that are not covered in this section, refer to "JEUS EJB Guide".

There are two security configuration methods as follows:

- [Configuring ejb-jar.xml](#)
- [Configuring jeus-ejb-dd.xml](#)

3.2.1. Configuring ejb-jar.xml

The following security items can be specified in ejb-jar.xml.

- Role-to-Role Reference mapping
- security identity configuration
- Logical role declaration
- EJB method permission configurations

The following example illustrates how to specify the aforementioned security items in ejb-jar.xml.

Security Configurations: <ejb-jar.xml>

```
<?xml version="1.0"?>
<ejb-jar xmlns="https://jakarta.ee/xml/ns/jakartaee">
  <display-name>product</display-name>
  <enterprise-beans>
    <entity>
      <ejb-name>product</ejb-name>
      . . .
      <security-role-ref>
        <role-name>cust</role-name>
        <role-link>customer</role-link>
      </security-role-ref>
      <security-identity>
        <run-as>
          <role-name>administrator</role-name>
        </run-as>
      </security-identity>
    </entity>
```

```

</enterprise-beans>
<assembly-descriptor>
    . . .
    <security-role>
        <role-name>administrator</role-name>
    </security-role>
    <security-role>
        <role-name>customer</role-name>
    </security-role>
    <method-permission>
        <role-name>administrator</role-name>
        <method>
            <ejb-name>product</ejb-name>
            <method-interfaces>Remote</method-interfaces>
            <method-name>getSecretKey</method-name>
            <method-params>
                <method-param>java.lang.Integer</method-param>
            </method-params>
        </method>
    </method-permission>
    <method-permission>
        <unchecked/>
        <method>
            <ejb-name>product</ejb-name>
            <method-name>doSomeAdmin</method-name>
            <method-params>
                <method-param>java.lang.String</method-param>
            </method-params>
        </method>
    </method-permission>
    <method-permission>
        <role-name>customer</role-name>
        <method>
            <ejb-name>product</ejb-name>
            <method-name>test1</method-name>
        </method>
    </method-permission>
    <exclude-list>
        <method>
            <ejb-name>product</ejb-name>
            <method-interfaces>Remote</method-interfaces>
            <method-name>getCustomerProfile</method-name>
            <method-params></method-params>
        </method>
    </exclude-list>
</assembly-descriptor>
</ejb-jar>

```

The following security items can be specified in ejb-jar.xml.

- **Role-to-Role Reference mapping**

The role-to-role reference mapping is specified in the `<security-role-ref>` tag. Its purpose is to map a role reference to a logical role. In the EJB code, the role reference is used instead of the actual Role.

The following tags are used inside the `<security-role-ref>` tag.

Tag	Description
<role-name>	The role reference which is used in EJB code instead of the actual role name.
<role-link>	The name of the actual role declared in the security-role tag. The role reference is mapped to a role indicated by the role-link.

• Security identity configuration

Every EJB in the module has a security identity, which will be propagated to other EJBs during remote calls. The security identity for an EJB is configured using the tags, <ejb-jar> <enterprise-beans> <type> <security-identity>.

The configurable child tags are as follows.

Tag	Description
<use-caller-identity>	If this tag is empty, the caller of the EJB will be used as the security identity.
<run-as>	If this tag is used, the principal of the role specified in <role-name> will be used as the security identity when an EJB is called. If the <run-as-identity> tag exists in jeus-ejb-dd.xml, this setting will be ignored.

• Logical role declaration

The <security-role> tag under the <assembly-descriptor> tag specifies the logical Roles that will apply to the corresponding EJB module. All role names in ejb-jar.xml must be specified in this tag.

• EJB method Permission configuration

The <method-permission> tag, inside the <assembly-descriptor> tag, specifies security constraints for each EJB method.

The following describes the child tags of the <method-permission>.

Tag	Description
<role-name> (optional)	Specifies the name of a logical role defined in <security-role>. It has access to the methods defined in the following.
<unchecked> (optional)	An empty tag. If specified, the following methods will be unchecked. Anyone can access these methods, regardless of its Role.

Tag	Description
<code><method></code> (one or more)	<p>The method to which the <code><role-name></code> and <code><unchecked></code> tags will be applied. The child tags are as follows.</p> <ul style="list-style-type: none"> • <code><ejb-name></code>: The EJB name that contains the method. • <code><method-intf></code>: The EJB interface type that contains the method. If omitted, all interfaces in local and remote servers are implied. • <code><method-name></code>: EJB method name. • <code><method-params></code> (optional): Can have multiple <code><method-param></code> tags. If the <code><method-param></code> tag is omitted, the same permissions will be applied to any method with the given method-name, regardless of the parameter. (In EJB, there may be multiple methods that have the same name but different parameters. The Method Permission will be applied to all the methods with the same name.) <p>The value of the <code><method-param></code> tag must be a fully qualified Java class name that includes the package name or Java Primitive type such as <code>int</code>.</p> <p>If <code><method-params></code> is empty (<code><method-params/></code>), the method has no parameters.</p>
<code><exclude-list></code> (one)	<p>Defines excluded EJB methods that no one can access. Only one <code><excluded-list></code> tag can exist, but can have multiple child <code><method></code> tags.</p>



Refer to [Role-to-Resource Mapping](#) for an explanation of the previous example. Refer to the EJB specification for more information about these tags. Also refer to "JEUS EJB Guide" for information about deploying EJB modules in JEUS.

3.2.2. Configuring jeus-ejb-dd.xml

The following security items can be configured in `jeus-ejb-dd.xml`.

- Security-related configurations
- Configuring EJB methods that are not defined in `ejb-jar.xml`
- Configuring EJB principal which uses the run-as identity

While the `ejb-jar.xml` file takes care of most role-to-method mappings for the EJB, it still needs to determine where to declare the principal-to-role mappings and how to handle EJB methods that were not handled in `ejb-jar.xml`. These mappings and handling are configured in the `jeus-ejb-dd.xml` file, which usually resides in the META-INF directory of the deployed EJB .jar archive.

The following example illustrates how to configure the previous security configuration items in `jeus-ejb-dd.xml`.


```

<?xml version="1.0"?>
<jeus-ejb-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <module-info>
    <role-permission>
      <principal>user1</principal>
      <role>administrator</role>
      <classname>mypackage.MyRolePermission</classname>
    </role-permission>
    <role-permission>
      <principal>user2</principal>
      <role>customer</role>
    </role-permission>
    <role-permission>
      <role>roleA</role>
      </excluded>
    </role-permission>
    <role-permission>
      <role>roleB</role>
      </unchecked>
    </role-permission>
    <unspecified-method-permission>
      <role>administrator</role>
    </unspecified-method-permission>
  </module-info>
  <beanlist>
    <jeus-bean>
      . . .
      <run-as-identity>
        <principal-name>user1</principal-name>
      </run-as-identity>
    </jeus-bean>
    . . .
  </beanlist>
</jeus-ejb-dd>

```

The following security items can be configured in jeus-ejb-dd.xml.

- **Security-related configurations**

Most security-related settings in jeus-ejb-dd.xml are set in the <jeus-ejb-dd> tag just below the parent tag, <module-info>.

- <role-permission> (0 or more)

Descriptions of each child tag of <role-permission> are as follows. Each tag specifies permission for a role (usually a principal-to-role mapping).

Tag	Description
<principal> (0 or more)	The principal name that is mapped to a Role.

Tag	Description
<role> (required)	Only one is allowed. A required name of the logical role. This name will be passed as the first parameter of the constructor of the Java class that implements the role permission. The role name must match the logical name specified in ejb-jar.xml.
<actions> (optional)	Contains additional data for the role permission. If set, this action data will be passed as the second parameter of the constructor of the Java class that implements the role permission.
<classname> (optional)	The fully qualified Java class name of the class that implements the role permission. This class must be a subclass of the java.security.Permission class and have a public constructor that accepts at least one parameter for the role name. If omitted, the jeus.security.resource.RolePermission class is used.
<excluded> (optional)	An empty tag. If set, the role will be excluded (no one can access the Role(s) defined for the role permission). This takes precedence over the <principal> and <unchecked> tags.
<unchecked> (optional)	An empty tag. If set, the role will be unchecked (anyone can access the Role(s) defined in the role permission). This takes precedence over the <principal> tag.

- **Configuring EJB methods that are not defined in ejb-jar.xml**

There are three ways to handle EJB methods that are not specified in ejb-jar.xml. They are defined in the child tag, **<unspecified-method-permission>**, inside the **<module-info>** tag. The configurable child tags are as follows.

Tag	Description
<role> (optional)	If set, all unspecified methods will be mapped to this Role.
<excluded> (optional)	If set, all unspecified methods will be excluded (accessible by no one). This takes precedence over the <role> and the <unchecked> tags.
<unchecked> (optional)	If set, all unspecified methods will be unchecked (accessible by anyone). This takes precedence over the <role> tag.

- **Configuring EJB Principal which uses the run-as identity**

The principal needs to be set for an EJB that uses the run-as identity. The principal name specified in the <jeus-bean> <run-as-identity> <principal-name> tag is used. The principal must be in the role that is specified in the <security-identity><run-as><role-name> tag of ejb-jar.xml.



1. Refer to [Role-to-Resource Mapping](#) for an explanation of the example.
2. For detailed information about each tag, refer to "JEUS EJB Guide".

3.3. Configuring Web Module Security

This section explains in detail how to configure security for web (Servlet) modules.

The security of web modules is divided into two parts:

- [Configuring web.xml](#): Role-to-Web Resource mapping configurations
- [Configuring jeus-web-dd.xml](#): principal-to-role mapping configurations

3.3.1. Configuring web.xml

web.xml is the main DD file for a Web archive (WAR file). This is a standard Jakarta EE DD file that usually resides in the WEB-INF directory of the WAR file.

The security items that can be configured in web.xml are as follows.

- **Run-as Identity**

Every Servlet in the Web module has a security identity, which will be propagated to EJBs during a remote call.

The security identity for a servlet is configured in the `<web-app><servlet><run-as>` tag, which has the following child tag.

Tag	Description
<code><role-name>(optional)</code>	The role that runs the servlet. If omitted, the caller identity will be used.

- **Role-to-role reference mapping**

The role-to-role reference mapping is specified in the `<security-role-ref>` tag. Its purpose is to map a role reference to a logical role. In the EJB code, the role reference is used instead of the actual Role.

The `<security-role-ref>` tag is configured inside the `<servlet>` tag of web.xml and has the following child tags.

Tag	Description
<code><role-name></code>	The role name that is used in Servlet codes.
<code><role-link></code>	The actual role name defined in the <code><security-role></code> tag. The role reference will be mapped to the actual role defined in the <code><role-name></code> tag.

- **Permissions to access servlet URL (Security constraints)**

To restrict the access to Servlet URL, the `<security-constraint>` tag inside the `<web-app>` tag is used.

`<security-constraint>` has the following child tags.

- `<web-resource-collection>` (one or more)

Specifies a list of Web resources for which access constraints are set.

Tag	Description
<code><web-resource-name></code>	The name of the web resource.
<code><url-patterns></code> (0 or more)	<p>The URL (relative to the context root) pattern that is a part of the Web resource.</p> <p>The following are URL pattern examples.</p> <ul style="list-style-type: none">◦ <code>/mywebapp/*</code>: All URL that includes mywebapp◦ <code>/something</code>: The exact URL◦ <code>*.jsp</code>: All resources that ends with ".jsp"
<code><http-method></code> (0 or more)	<p>The HTTP methods that apply to the web resources.</p> <ul style="list-style-type: none">◦ GET◦ POST◦ PUT

- `<auth-constraint>` (optional)

Specifies the roles that can access the web resource defined in the `<security-constraint>` tag. There must be zero or more **`<role-name>`** child tag(s), each specifying a logical role name that allows access to the resource.

If no role name is specified and it is empty (`<auth-constraint/>`), no one can access the web resource (same as excluded Web resources).

If the `<auth-constraint>` tag is omitted, the web resource will be unchecked so that any one can access it regardless of the role.

- `<user-data-constraint>` (optional)

Sets whether to declare a "transport guarantee" for connections that are made to the web resources.

It has the `<transport-guarantee>` child tag, which specifies the guarantee level for the Web resource connections.

Value	Description
NONE	The connection is unprotected.
INTEGRAL	The connection guarantees the message integrity (ensures that the sent message has not been modified).
CONFIDENTIAL	Ensures that the messages sent are encrypted to prevent eavesdropping.

• Logical Role Declaration

After the `<security-constraint>` tag, there is the `<security-role>` tag that declares the logical roles. Each tag has the `<role-name>` tag, that declares the logical role names defined in DD.

The following is an example of security configurations of `web.xml`.

Web Module Security Configurations: `<web.xml>`

```
<?xml version="1.0"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee">
  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    . . .
    <run-as> ①
      <role-name>R1</role-name>
    </run-as>
    <security-role-ref> ②
      <role-name>adminRef</role-name>
      <role-link>R1</role-link>
    </security-role-ref>
    . . .
  </servlet>
  <security-constraint> ③
    <web-resource-collection>
      <web-resource-name>A</web-resource-name>
      <url-pattern>/a/*</url-pattern>
      <url-pattern>/b/*</url-pattern>
      <url-pattern>/a/</url-pattern>
      <url-pattern>/b/</url-pattern>
      <http-method>DELETE</http-method>
      <http-method>PUT</http-method>
    </web-resource-collection>
    <web-resource-collection>
      <web-resource-name>B</web-resource-name>
      <url-pattern>*.asp</url-pattern>
    </web-resource-collection>
    <auth-constraint/>
  </security-constraint>
  <security-constraint> ④
    <web-resource-collection>
      <web-resource-name>C</web-resource-name>
      <url-pattern>/a/*</url-pattern>
      <url-pattern>/b/*</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
    <web-resource-collection>
      <web-resource-name>D</web-resource-name>
```

```

        <url-pattern>/b/*</url-pattern>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>R1</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>
            CONFIDENTIAL
        </transport-guarantee>
    </user-data-constraint>
</security-constraint>
<security-role> ⑤
    <role-name>R1</role-name>
</security-role>
<security-role>
    <role-name>R2</role-name>
</security-role>
<security-role>
    <role-name>R3</role-name>
</security-role>
</web-app>

```

The following describes the previous example:

- ① The run-as-identity is set to role R1, meaning that the propagated security identity of the servlet must be a principal that is in R1. The actual principal is specified in the JEUS DD file.
- ② In the <security-role-ref> tag, the actual role R1 is mapped to the role reference adminRef. In the actual Servlet source codes, the role reference adminRef is used instead of the Role.
- ③ The first security constraint specifies that the URLs with patterns, "/a", "/b", "/a/*", "/b/*" and HTTP methods DELETE and PUT, and all URLs that end with "*.asp" must NOT be accessible to anyone. This is because it is set to <auth-constraint/>.
- ④ The second security constraint says that the URLs with patterns, "/a/*", "/b/*" and HTTP method GET, and all URLs with the pattern /b/* and HTTP method POST should only be accessible by Principals that are in the role R1 and the connection should be CONFIDENTIAL.
- ⑤ Declares three logical roles: R1, R2, and R3.

If a web resource (URL pattern + HTTP method) is not set separately in web.xml, it means that anyone can access the resource regardless of role. This is the default setting for web application access in Jakarta EE.



For detailed information on all tags in web.xml, refer to the servlet specifications. Also, for additional information on deploying servlets in JEUS, refer to "JEUS Web Engine Guide".

3.3.2. Configuring jeus-web-dd.xml

This section explains how to configure jeus-web-dd.xml, which is the JEUS DD file for web modules.

- **Defining Principal-to-Role mapping**

To define a principal-to-role mapping in jeus-web-dd.xml, use the `<jeus-web-dd><role-mapping>` tag.

The following are child tags of `<role-mapping>`.

- `<role-permission>` (0 or more)

Specifies permission for a role (usually a principal-to-role mapping).

Tag	Description
<code><principal></code> (0 or more)	The principal name that is mapped to a Role.
<code><role></code> (1, required)	The logical role name. This will be passed as the first parameter of the constructor of the Java class that implements the role permission. The role name must match the logical role name in web.xml.
<code><actions></code> (optional)	Contains additional data for the role permission. If set, this action data will be passed as the second parameter of the constructor of the Java class that implements the role permission.
<code><classname></code> (optional)	<p>The Java class name that implements the role permission.</p> <p>This class must be a sub class of the <code>java.security.Permission</code> class and have a public constructor that accepts at least one string parameter (the role name).</p> <p>If omitted, the <code>jeus.security.resource.RolePermission</code> class is used.</p>
<code><excluded></code> (optional)	<p>An empty tag. If set, the role will be excluded. In other words, no principal can be in the role.</p> <p>This takes precedence over the <code><principal></code> and <code><unchecked></code> tags.</p>
<code><unchecked></code> (optional)	<p>An empty tag. If set, the role will be unchecked. In other words, any principal can be in the role.</p> <p>This takes precedence over the <code><principal></code> tag.</p>

- **Setting `<run-as-principal>` in servlet**

To set `<run-as-principal>` tag in a servlet, define the `<servlet>` tag and add the `<run-as><principal-name>` tag to it. The value of the `<principal-name>` tag must be the principal that belongs to the role defined in `<run-as><role-name>` tag of web.xml.



Currently, there are no tags to specify how to handle the Servlet URLs that are not defined in web.xml of jeus-web-dd.xml. In the Servlet specifications, all unspecified Servlet URLs are always regarded as unchecked and accessible by everyone.

The following is an example of security configurations in jeus-web-dd.xml.

Web Module Security Configurations: <jeus-web-dd.xml>

```
<?xml version="1.0"?>
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <context-path>/tutorial</context-path>
  <docbase>Tutorial</docbase>
  . . .
  <role-mapping>
    <role-permission> ①
      <principal>user1</principal>
      <principal>user2</principal>
      <principal>customerGroup</principal>
      <role>R1</role>
    </role-permission>
    <role-permission> ②
      <role>R2</role>
      </excluded>
    </role-permission>
    <role-permission> ③
      <role>R3</role>
      </unchecked>
    </role-permission>
    <role-permission> ④
      <principal>tellerGroup</principal>
      <role>R4</role>
      <actions>09:00-17:00</actions>
      <classname>
        jeus.security.resource.TimeConstrainedRolePermission
      </classname>
    </role-permission>
  </role-mapping>
  <servlet> ⑤
    <servlet-name>HelloWorld</servlet-name>
    <run-as-identity>
      <principal-name>user1</principal-name>
    </run-as-identity>
    . . .
  </servlet>
  . . .
</jeus-web-dd>
```

The following describes the previous example:

- ① The Principals user1, user2, and customerGroup are in the role R1.
- ② The role R2 is excluded so no one will be in the role.

- ③ The role R3 is unchecked so any one can be in the role.
- ④ The principal tellerGroup will be in the role R4 on the condition that the current time is between 9 am and 5 pm. This logic is implemented in the `implies()` method of the `jeus.security.resource.TimeConstrainedRolePermission` class.
- ⑤ `user1` is used as a run-as-principal. When the Servlet calls an EJB, `user1` will be passed as the security identity. Note that the deployer must check that `user1` is in the role R1, which is defined in `web.xml`.

3.4. Configuring Jakarta EE Application Security

This section explains how to set security for Jakarta EE applications (EAR file).

The basic procedure is as follows:

- [Configuring application.xml](#): Declare logical roles.
- [Configuring jeus-application-dd.xml](#): Configure the security domain for deployment and principal-to-role mapping.

3.4.1. Configuring application.xml

Jakarta EE application is an archive file with the ".ear" extension. The EAR archive consists of EJB, Web, and connector modules, and any other necessary supporting classes. The META-INF directory contains the `application.xml` configuration file that provides various information about the application.

This section explains the security-related information in `application.xml`.

There is basically only one type of security information in `application.xml`: declaration of logical security Roles. The role declaration applies to all EJB and Web modules in the EAR file and has an application scope.

Jakarta EE Application Security Configuration: `<application.xml>`

```
<?xml version="1.0" encoding="UTF-8"?>
<application version="1.4" . . .>
  <description>Application description</description>
  <display-name>myApp</display-name>
  <module>
    <web>
      <web-uri>myWebApp.war</web-uri>
      <context-root>myWebApp</context-root>
    </web>
  </module>
  <module>
    <ejb>myEjbApp.jar</ejb>
  </module>
  <security-role> ①
    <role-name>Administrator</role-name>
  </security-role>
```

```
<security-role> ②
  <role-name>Customer</role-name>
</security-role>
</application>
```

In the previous example, the following two roles are declared.

- ① "Administrator"
- ② "Customer"

These two roles are used in the DD files of EJB and Web modules to define role-to-resource mappings. Securities for EJB and web modules are explained in [Configuring EJB Module Security](#) and [Configuring Web Module Security](#).

Before deploying applications to the JEUS Server, ensure that logical roles are mapped to the actual Principals. This mapping is configured in the following files.

- For applications: jeus-application-dd.xml
- For EJB modules: jeus-ejb-dd.xml (for more information about principal-to-role mapping, refer to [Configuring jeus-ejb-dd.xml](#))
- For Web modules: jeus-web-dd.xml (for more information about principal-to-role mapping, refer to [Configuring jeus-web-dd.xml](#))

It is important to understand all the modules in a Jakarta EE application, because Roles and principal-to-role mappings are shared within the limits of the application. If the role Administrator is defined in the application.xml file, and the principal user2 is granted the role Administrator in jeus-application-dd.xml, this role will be applied to all EJBs and Web modules in the application.

In the same way, even though this role and principal-to-role mapping are defined only in jeus-ejb-dd.xml of a particular EJB module, they are accessible to all modules in the EAR file.



Note that all principal-to-role mappings in Jakarta EE applications have an application scope whether it is configured in jeus-application-dd.xml, or in a certain jeus-ejb-dd.xml or jeus-web-dd.xml file.

3.4.2. Configuring jeus-application-dd.xml

The principal-to-role mappings on the application level are set in **jeus-application-dd.xml**.

To define a principal-to-role mapping, add the following tags under the <application> tag in jeus-application-dd.xml.

- <role-permission> (optional)

Each <role-permission> tag defines a principal-to-role mapping and can optionally configure Excluded and Unchecked Roles. This tag works exactly the same that in jeus-web-dd.xml, jeus-ejb-

dd.xml, and policies.xml files.

It has the following child tags.

Tag	Description
<principal> (0 or more)	The principal name that is mapped to a Role.
<role> (1, required)	<p>The logical role name. This will be passed as the first parameter of the constructor of the Java class that implements the role permission.</p> <p>The role name must match the logical role name in application.xml.</p>
<actions> (optional)	Contains additional data for the role permission. If set, this action data will be passed as the second parameter of the constructor of the Java class that implements the role permission.
<classname> (optional)	<p>The name of the Java class that implements the role permission. This class must be a subclass of the java.security.Permission class and have a public constructor that accepts at least one string parameter (the role name).</p> <p>If omitted, the jeus.security.resource.RolePermission class is used.</p>
<excluded> (optional)	<p>An empty tag. If set, the role will be excluded. No one can be in the Role(s) defined in the role permission.</p> <p>This takes precedence over the <principal> and <unchecked> tags.</p>
<unchecked> (optional)	<p>An empty tag. If set, the role will be unchecked. Anyone can be in the Role(s) defined in the role permission.</p> <p>This takes precedence over the <principal> tag.</p>

The security domain to which the application is deployed must be specified inside the <application> tag. By default, the security domain, SYSTEM_DOMAIN, will be used, but if a particular application requires a specific security service, a new domain that includes the security service must be created and deployed.

The security domain name can be specified using the **deploy** command of jeusadmin.

Jakarta EE Application Security Configurations: <jeus-application-dd.xml>

```
<application>
  . . .
  <role-permission> ①
    <principal>user2</principal>
    <role>Administrator</role>
  </role-permission>
  <role-permission> ②
    <role>Customer</role>
    </unchecked>
  </role-permission>
  . . .
</application>
```

In the previous example, the Jakarta EE application security is set as follows:

- ① The principal user2 is mapped to the role Administrator, and uses the default RolePermission.
- ② Since the role is unchecked, all principals are in the role Customer.



1. For more information about configuring a security domain, refer to [Configuring Security Domain](#).
2. For more information about how to develop and configure Custom Permissions, refer to [Configuring Security System Policy](#).
3. For information about how to deploy a Jakarta EE application on JEUS, refer to "JEUS Server Guide".

3.5. Example

This section briefly introduces the configuration examples from the previous sections.

The following page shows the login page for accessing the main page. The following page can be accessed only when the checked permission has been set for the URL. If the unchecked permission is set, the page will directly go to the main page. If the excluded permission is set, authorization will fail and an error page will appear.

JEUSTM Samples
Web Application Server

[home](#)

Login

Welcome! This page gives information of the security configuration for applications (EJBs and Servlets). Please login to access the main page.

Username:	<input type="text"/>
Password:	<input type="password"/>
	<input type="button" value="Login"/>

Login Page

If authorization fails or an attempt was made to login using an unauthorized account, an error page will appear. If the login is successful, the following page will appear.

[home](#)

JEUS Security Sample

The following samples are offered to show how the security configuration works. Each sample's target resource is a Web URL or an EJB method.

Servlet Sample

servlet-name	authorization	servlet-class
unchecked	unchecked	webpages.JoinUs
checked	checked	webpages.OnlyOne
excluded	excluded	webpages.WithYou

EJB Sample

method-name	authorization	ejb-name
message	unchecked	ejbwork.PublicCall
message	checked	ejbwork.Admin
excludedMSG	excluded	ejbwork.Secret
callsecret	applied run-as	ejbwork.PublicCall

Main Page

The previous main page has three Servlet links, and can call three EJB methods. A successful login is dependent on the login account and the configuration files. Thus, a user can directly check for accessibility using different configurations and login accounts.

4. Programming with the Security System API

This chapter explains about programming with the security system API.

4.1. Overview

This chapter explains how to develop programs with the security system API to add user's own security features to user applications. For example, there is the registration Servlet (called "auto-registration") that automatically registers a new user to the JEUS security system by using an application.

Before developing a security service, application programmers should check whether the standard Jakarta EE security model and the JEUS security services support the desired security features. Developing a program using the security API will decrease the compatibility between Jakarta EE servers. It is recommended to use only the standard Jakarta EE security interfaces to maintain the compatibility.

4.2. Configuring Java SE Permissions

The security system API can be used to protect the JEUS system from malicious user code (Servlet or EJB) injections.

The security API can be used for user codes (Servlet or EJB) in the case when the Java security manager is used, or when the source code (Servlet, EJB) successfully logs in using the `LoginService.login (Subject)` method. In this case, the Subject is the subject of the user defined in the `accounts.xml` file in the target security domain and has the necessary JEUS Permission configured in the `policies.xml` file.

For more information about how to configure each file, refer to the following:

- Java SE security manager and Java SE Policy files: [Configuring Java SE SecurityManager](#)
- `accounts.xml`: [Configuring Security System User Information](#)
- `policies.xml` in the JEUS security system: [Configuring Security System Policy](#) and [References](#)

4.3. Basic API

The following classes in the `jeus.security.base` package play an important role when working with the security system at the application programming level.

Class	Description
jeus.security.base.Subject	<p>A user.</p> <p>A Subject has a single main principal, which acts as the Subject ID (username). Several string property values may be sent to the jeus.security.base.Subject class.</p>
jeus.security.base.CredentialFactory	<p>The member variable of the Subject class. It is used to create the actual credentials of a Subject.</p> <p>For example, the PasswordFactory class creates a password credential instance, and the JKSCertificateFactory class gets a certificate from JKS keystore and creates a credential instance for a certificate.</p>
jeus.security.base.Policy	Represents a single principal-to-role mapping and several role-to-resource mappings. Contains PermissionMaps as a member variable.
jeus.security.base.PermissionMap	The container for java.security.Permission instances and the member variables of the Policy class.
jeus.security.base.Role	<p>The interface that represents a logical role.</p> <p>In role-to-resource PermissionMap, the role instance is mapped to the resource permission. In the same way, in principal-to-role PermissionMap, the principal is mapped to role permission.</p>
jeus.security.base.SecurityCommonService	Authenticates the subject and checks the permission for the Subject.
jeus.security.base.SecurityException	The exception that occurs due to a security violation, such as failed login, failed authentication, and failed authorization.
jeus.security.base.ServiceException	The exception that occurs due to a critical runtime error in the security system.

4.4. Resource API

As well as the basic classes in the jeus.security.base package, the classes in the jeus.security.resource package also play an important role related to resources.

Class	Description
jeus.security.resource.PrincipalImpl	The implementation class of the java.security.Principal interface.
jeus.security.resource.GroupPrincipalImpl	The subclass of PrincipalImpl that represents group principals and the implementation class of the java.security.acl.Group that manages members of the related nested groups.
jeus.security.resource.Password	A simple Password Credential instance. It is created by the PasswordFactory.

Class	Description
jeus.security.resource.PasswordFactory	The CredentialFactory that creates the Password Credential.
jeus.security.resource.Lock	A credential that locks the relevant subject. Any attempt to login to the locked subject will always fail.
jeus.security.resource.LockFactory	The CredentialFactory that creates the lock credential.
jeus.security.resource.ExpiryTime	A credential. Sets the expiration of the subject. Any attempt to login with the subject, that has already expired, will always fail.
jeus.security.resource.ExpiryTimeFactory	The CredentialFactory that creates the ExpiryTime.
jeus.security.resource.RoleImpl	The class that implements the role interface.
jeus.security.resource.RolePermission	The subclass of java.security.Permission that represents a particular principal belonging to a particular role. The RolePermission is used to express principal-to-role mapping.
jeus.security.resource.TimeConstrainedRolePermission	A sub class of RolePermission that represents a principal being in the role expressed by the its super class only during the specified time period.
jeus.security.resource.ResourcePermission	The subclass of java.security.Permission that expresses the concept that a role can access a resource and perform particular actions. ResourcePermission is used to express role-to-resource mapping.



For more information about the classes, refer to the Javadoc and [References](#).

4.5. SPI Class

To communicate with the services that are the foundation of the security system, use the following SPI classes from the jeus.security.spi package.

Class	Description
jeus.security.spi.AuthenticationRepositoryService	Used to add, remove, and search for Subject to/from a Subject repository. A Subject (user) can be added within the program using this class.
jeus.security.spi.AuthorizationRepositoryService	Used to add, remove, and search for Policy data to/from a Policy repository. Permission can be added within the program using this class.



Refer to the Javadoc and [References](#) for more information. Also refer to [Developing Custom Security Service](#) for more information about these SPI classes.

4.6. Example

The following shows how to develop a program with the security API.

```
// Login the CodeSubject so that security checks are
// disabled (so that we can modify the Subject and Policy
// stores)
SecurityCommonService.loginCodeSubject();

// Make Subject with Principal "pete"
Principal petePrincipal = new PrincipalImpl("pete");
Subject pete = new Subject(petePrincipal);

// Make password "petepw" for Subject "pete"
PasswordFactory pf = new PasswordFactory("petepw");
pete.getCredentialFactories().add(pf);

// Add new Subject to the Subject store
AuthenticationRepositoryService.addSubject(pete);

// Make a new Policy
Policy policy = new Policy();

// Make role "someRole"
Role someRole = new RoleImpl("someRole");

// Make a RolePermission for role "someRole"
Permission rolePermission = new RolePermission(someRole);

// Add the RolePermission for "someRole" to the Policy
policy.getRolePolicy().addPermission(
    rolePermission, new Object[] {petePrincipal}, false, false);

// Create a ResourcePermission for resource "rsc1" with actions
// "action1" and "action2"
Permission rscPermission =
    new ResourcePermission("rsc1", "action1,action2");

// Add the ResourcePermission to the Policy using
// context id "ctx1"
policy.getResourcePolicy("ctx1", true).addPermission(
    rscPermission, new Object[] {someRole}, false, false);

// Add the new Policy to the Policy store
AuthorizationRepositoryService.addPolicy(policy);

// Logout the CodeSubject so that security checks are
// enabled again
SecurityCommonService.logout();
```

```
// Make a Subject to be logged in
Subject pete2 = Subject.makeSubject("pete", "petepw");

// Login Subject "pete" (should succeed since we added
// "pete" earlier)
SecurityCommonService.loginDefault(pete2);

// Check ResourcePermission "rsc1" for current Subject ("pete")
// Should succeed since we added Policy for this above
SecurityCommonService.checkPermission(
    "ctx1", new ResourcePermissin("rsc1", "action2");

// Print the name of the current Subject ("pete")
System.out.println(
    SecurityCommonService.getCurrentSubject().getPrincipal().getName());

// Logout "pete"
SecurityCommonService.logout();
```

5. Developing Customized Security Services

This chapter describes how to develop custom security services, which is a key feature of the JEUS security system.

5.1. Overview

Using this feature, you can easily integrate the JEUS security system with various kinds of external security systems or security data repositories.

The following sections describe how to develop custom security services.

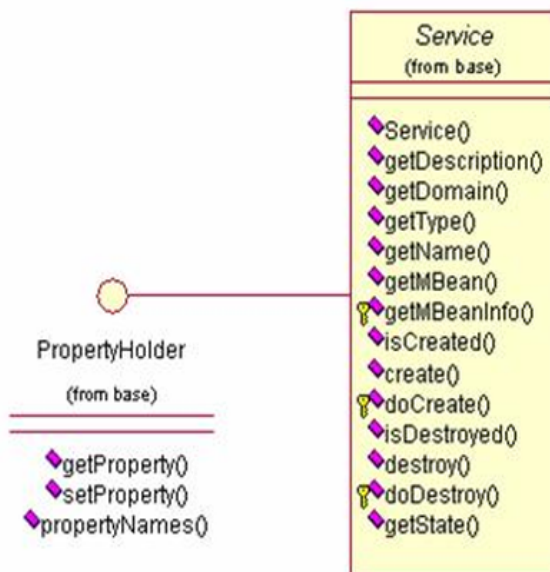
- [Service Class](#)
- [Implementation Basic Pattern](#)
- [11 SPI Classes](#)
- [Custom Security Service Configurations](#)

5.2. Service Class

The most basic class of the pluggable security architecture is the class `jeus.security.base.Service` class.

The Service class is an abstract class that must be extended by all security service implementation classes. Currently, all available security SPI classes in the `jeus.security.spi` package extend this class.

The following is the class diagram for the Service class.



Service Class Diagram

The Service class provides a few basic things that apply to all security services.

- **Service Description**

A Service class provides a string type description that briefly explains what the service does. The description can be obtained using the **getDescription()** method.

- **Service Domain**

Each Service instance will be assigned to a domain when the service instance is created at runtime. So a domain will be essentially a collection of service instances. The domain of the service can be obtained using the **getDomain()** method.

- **Service Type**

Each Service implementation class has a single type. The type acts like a marker so that a specific service instance can be obtained from a set of service instances.

For example, if a Service implementation class implements some authentication functions, other security services need to request the domain for the class in order to use this authentication service class. To do so, they can pass the authentication type of the class so that the domain can return the correct service instance.

The service type can be obtained by using the **getType()** method. However, since **getType()** is an abstract method, it must be implemented by the sub-classes. The service type is actually a `java.lang.Class`, an instance of the SPI class that is included in the `jeus.security.spi` package. Each SPI class implements the `getType()` method as final. That is, the `getType()` method cannot be redefined in a sub-class of SPI.

- **Service**

The service name can be obtained using the **getName()** method.

- **Service MBean**

The service class is associated with a JMX MBean, which is used to manage a particular service instance.

The MBean can be obtained using the **getMBean()** method. Sub-classes must override this method in order to return a different MBean instead of the default one.

By default, MBean is created based on the information that is returned from the `getMBeanInfo()` method that is declared as a protected method. Sub-classes can redefine this method to return a different `MBeanInfo` instance instead of the default one. To do so, the overriding implementation must include the `MBeanInfo` of the super class.

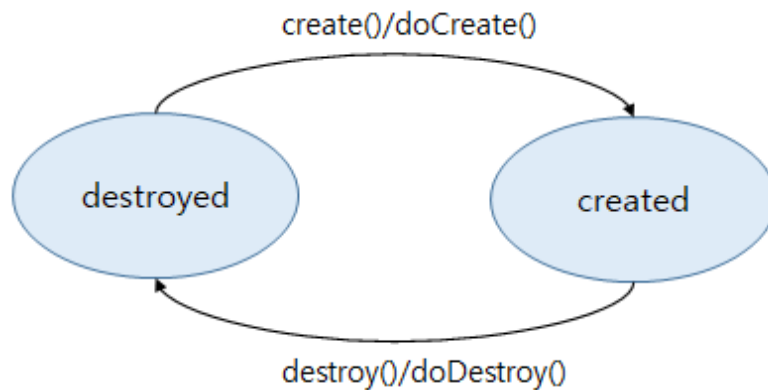
- **Service State**

Each Service implementation class is in one of the two states, created or destroyed. The state can be changed by invoking `create()` and `destroy()`. These methods delegate the actual service to the abstract protected methods, **doCreate()** and **doDestroy()**. It is required that all service implementation classes must implement both **doCreate()** and **doDestroy()** methods.

The `doCreate()` and `doDestroy()` methods include the code that starts and terminates the service. The typical `doCreate()` method contains the code that acquires some resources like DB connections, while the `doDestroy()` method contains the code that releases those resources.

The current status of the service class can be obtained using the `isCreate()` and `isDestroyed()` methods. There is also the method `getState()` that returns the current status as a string.

The following is the status-chart for the Service class.



The State-Chart of the Service Class

• Service Properties

The service class has a name-value pair property. This property can be set or retrieved using the `jeus.security.base.PropertyHolder` interface. The service implementation class must implement the interface.

The properties are used to initialize service instances. The properties are set before a service instance is created, and then the `create()` method is called. After the properties have been set, the service instances can be initialized by calling the `doCreate()` method to retrieve the property values.

5.3. The Basic Pattern of Implementing Custom Security Services

This section briefly describes how to implement the custom security services.

The following are required to implement a customized security service.

1. Users must have a general understanding of the security system and its architecture.
2. Choose the security functions that the custom security service needs to provide.
3. Choose the SPI class that has the necessary features. For more information about main SPI classes, refer to [SPI Class](#).
4. Refer to the documents for information on SPI class. (Refer to Javadoc, [References](#), and [SPI Class](#).)
5. Make a sub-class of the selected SPI class and implement the following methods. An empty public constructor without parameter must be provided.

- a. Optionally define a set of properties that are used to initialize services. Each property is a public static final string type and what each property represents must be documented.
 - b. Implement the `doCreate()` method. This method will be called once when the security service is started and it performs general initialization operations such as resource allocation. This method reads the property values using the `getProperty()` method. The parameter of the `getProperty()` method is the property name that was defined in the previous step.
 - c. Implement the `doDestroy()` method. This method is called once when the service is terminated. The method should release any resources that were allocated during the `doCreate()` method call and also perform clean-up operations such as writing logs to a certain file.
 - d. Implement all abstract methods in the selected SPI class as specified in the Javadoc.
 - e. Optionally implement the methods to be used for managing the service through JMX. Implement the `getMBean()` or `getMBeanInfo()` method.
6. Compile the class that implements SPI.
 7. Register the new security service with JEUS as described in [Configuring Security System](#).

5.4. SPI Class

The following SPI classes are defined in the `jeus.security.spi` package.

The following is the list of SPI classes defined in the `jeus.security.spi` package. The cardinality indicates how many SPI instances can exist for each security domain. For the detailed information about SPI classes, refer to [References](#).

class name	Purpose	Cardinality
<code>SecurityInstaller</code>	Installs and uninstalls the security system. Only one must exist for the entire JVM.	1
SubjectValidationService	Checks if the credential of the subject is valid prior to logging in.	0 or more
SubjectFactoryService	Creates the custom subject prior to logging in.	1
AuthenticationService	Authenticates the subject prior to logging in.	1
AuthenticationRepositoryService	Add, get, or remove a subject from its repository.	1
IdentityAssertionService	Maps credentials to subject using the <code>cert-user-map.xml</code> file.	0 or more
CredentialMappingService	Maps credentials to subject using the truststore information.	0 or more
CredentialVerificationService	Verifies that at least one of the credentials for the subject is valid.	0 or more

class name	Purpose	Cardinality
AuthorizationService	Checks whether the subject has permission to access a role or resource.	1
AuthorizationRepositoryService	Add, get, or remove Policies from the repository.	1
EventHandlingService	Implements custom event handler for security events and various security audits.	0 or more

5.4.1. SubjectValidationService SPI

The `jeus.security.spi.SubjectValidationService` SPI is used to check whether all the credentials held by a subject are valid. This means that there should be no invalid credentials for any reason. An invalid credential means that the Subject is invalid and consequently that the subject should not be allowed to log in.

A typical use for the `SubjectValidationService` instance is to check whether the subject contains a lock credential. If so, it can be concluded that the subject is locked out and that any login process should not be allowed to proceed.

Thus, the `SubjectValidationService.checkValidity(Subject)` method is usually called during the login process. If a `SecurityException` occurs, the login process will fail. `EventHandlingService` class is used to automatically configure a lock credential for the subject. For more information, refer to [EventHandlingService SPI](#).

Note that the issue of authentication (verifying whether the returned subject corresponds to the actual subject) is different from the validation of the Subject. Authentication and validation are not dependent upon one another. However, login is dependent on both, as already mentioned.

ZERO OR MORE `SubjectValidationService` instances can be configured for each domain. If no `SecurityException` occurs during the `SubjectValidationService` process, the subject is regarded as valid, and the login process is allowed to proceed. However, if at least one `SecurityException` occurs, the overall validation will fail and the login must NOT be allowed to proceed.



`SubjectValidationService` SPI class is functionally different from the `CredentialVerificationService` class.

`SubjectValidationService` checks the credential to determine the validity of the subject. `CredentialVerificationService` checks if the subject has at least one credential to prove its authenticity. So the `SubjectValidationService` class is used after the subject has been successfully authenticated, while `CredentialVerificationService` class is used during the process of authentication in the `AuthenticationService` class.

5.4.2. SubjectFactoryService SPI

`jeus.security.spi.SubjectFactoryService` SPI is a special SPI class used to create a subject without any external information.

`SubjectFactoryService` SPI is used to create a subject using the `SubjectFactoryService` class without receiving any parameters. On the other hand, regular implementation classes receive the username and password from the prompt and create a subject based on the information.

The `SubjectFactoryService` class is created in the login mechanism to support other credential types other than the password.

5.4.3. AuthenticationService SPI

The `jeus.security.spi.AuthenticationService` class is used to authenticate a Subject. It is used to verify that the subject, which was received as a parameter, indeed corresponds to the subject that is registered with the subject repository.

The following is the authentication procedure:

1. Calls the `jeus.security.spi.CredentialMappingService.getSubjectName(Object)` in the login mechanism to check for the user the credential of the subject is mapped to.
2. Calls the `jeus.security.spi.AuthenticationRepository.getSubject(String username)` method to copy the registered subject from the subject repository to the local server. This is called a local subject.
3. If the local subject is not null, check whether the credential of the local subject matches that of the subject that needs to be authenticated. The comparison can be done by calling the SPI method, `jeus.security.spi.CredentialVerificationService.verifyCredentials(Subject, Subject)`.

In some cases, `equals(Object)` method is used to compare the credentials, but it lacks flexibility.

4. Finally, if all of the previous steps were completed successfully, the local Subject is returned from the `authenticate(Subject)` method and eventually, that Subject will be used to log into the `SecurityCommonService`.

One or more `AuthenticationService` instances can be configured for each domain. If at least one of the configured `AuthenticationService` instances is successfully authenticated, the entire `AuthenticationService` will regard the authentication as successful. If one instance successfully authenticates the subject, no additional `AuthenticationService` will be queried.

5.4.4. AuthenticationRepositoryService SPI

`jeus.security.spi.AuthenticationRepositoryService` SPI has methods that add a subject to the repository and delete or get a subject from the repository. This SPI is used in the classes that implement the `AuthenticationService` class, or it can be used directly within the Jakarta EE application code.

For example, this SPI must be used to write code that automatically adds a subject to the subject

repository whenever a new user registers through a web site.

The main purpose of the `AuthenticationRepositoryService` class is to store a `jeus.security.base.Subject` instance in a specified repository type at runtime. The typical repository types are XML file and database.

The implementation and use of `AuthenticationRepositoryService` SPI is optional. However, it must be provided in the case when the default `AuthenticationService` is used (since it calls the `AuthenticationRepositoryService.getSubject(String)` method during the authentication), or if it is directly used in a Jakarta EE application code.

Only one `AuthenticationRepositoryService` instance can be specified for a domain. Currently, multiple `AuthenticationRepositoryService`'s cannot be used.

5.4.5. IdentityAssertionService SPI

`jeus.security.spi.IdentityAssertionService` SPI is necessary in the case when the username of a subject is unknown, which means that the main principal is set to null. In this case, it needs to extract the credentials of the subject and try to match these credentials to a username based on the information in `cert-user-map.xml`. This username is later used to perform authentication.

A typical example is the use of a `java.security.Certificate` instance. The instance uses a certificate credential without the main principal. In such case, during the authentication, it receives the certificate credential from the subject and passes it as a parameter of the `IdentityAssertionService.getIdentity(object)` method.

The method calls the `IdentityAssertionService.doIdentity(object)` method again. In the method, the user name will be traced back using the attribute key value of the certificate defined in the `cert-user-map.xml` file and sent as a return value.

5.4.6. CredentialMappingService SPI

`jeus.security.spi.CredentialMappingService` is necessary in the case when the username of a subject is unknown, which means that the main principal is set to null. In this case, it needs to extract the credentials of the subject and try to match these credentials to a username, which may be used later to perform authentication.

A typical example is the use of a `java.security.Certificate` instance. The instance uses certificate credential without the main principal. In such case, during the authentication, it receives the certificate credential from the subject and passes it as a parameter of the `CredentialMappingService.getSubjectName(object)` method.

The method calls the `CredentialMappingService.doGetSubjectName(object)` method again. In the method, the user name will be traced back using the attribute key value of the certificate defined in the `cert-user-map.xml` file and sent as a return value.

5.4.7. CredentialVerificationService SPI

`jeus.security.spi.CredentialVerificationService` SPI is used to support a new type of Proof Credential.

Proof Credential is a kind of credential that may be used to prove the authenticity of the Subject that is returned as a parameter, which means that it corresponds to an existing subject. A typical example of the proof credential is a password that is implemented by the `jeus.security.resource.Password` class.

`CredentialVerificationService` SPI essentially declares the `doVerifyCredentials(subject, subject)` method, which is one method that must be implemented by its sub-class. The first subject in the signature of this method is the reference Subject, which contains the credentials of the actual subject that is registered with the subject repository. The second argument is the proof Subject, which contains the proof credentials. The `doVerifyCredentials(Subject, Subject)` method compares the credentials of the two subjects and checks for a match.

The matching between credentials may be done in a number of different ways (just using the `equals(object)` method may not be enough). If the credentials of any two subjects match, the method will return, otherwise, a `jeus.security.base.SecurityException` will occur.



In some cases, it is not necessary to actually use the information of the reference subject. Certain proof credentials can be checked by using only the information of the proof Subject, such as a Certificate Credential.

The `CredentialVerificationService` that matches with the `jeus.security.resource.Password` is the `jeus.security.impl.verification.PasswordVerificationService` class.

Zero or more `CredentialVerificationService`'s can be configured for each domain. If at least one match is found in the `CredentialVerificationService`, the entire `CredentialVerificationService` verification is regarded as a success. Otherwise, a `SecurityException` occurs and the entire verification is regarded as a failure, and the authentication will fail.

5.4.8. AuthorizationService SPI

`jeus.security.spi.AuthorizationService` SPI defines the methods for authorization of the security system. The purpose of this SPI is to answer the question, "Does the Subject S have the permission to perform the action A?".

A typical implementation class calls the `jeus.security.spi.AuthorizationRepositoryService.getPolicy(contextId)` method and analyzes the returned `jeus.security.base.Policy` object to obtain the answer to the previous question. It is also possible to use other implementations that do not use the `AuthorizationRepositoryService` SPI.

One or more `AuthorizationService` can be configured for each domain. If at least one `AuthorizationService` returns a positive value, the entire authorization will be regarded as a success. Additional `AuthorizationService` objects will not receive the permission query if one successfully authorizes the permission.

5.4.9. AuthorizationRepositoryService SPI

`jeus.security.spi.AuthorizationRepositoryService` SPI is used to add, remove, or get a `jeus.security.base.Policy` (which represents authorization policy) instance to/from the Policy repository. This SPI may be used in the `AuthorizationService` implementation classes. It can also be used directly in Jakarta EE application code to directly add or remove Policy to/from the Policy repository when a particular event occurs.

The main purpose of `AuthorizationRepositoryService` is to store a `jeus.security.base.Policy` instance to a certain repository type at runtime. The typical repository types are XML file, database, and LDAP server.

The implementation and use of `AuthorizationRepositoryServiceSPI` is optional, but it must be provided in the case when the default `AuthorizationService` is used (since it calls the `AuthorizationRepositoryService.getPolicy(String)` method during authentication), or if it is directly used in Jakarta EE application code.

Only one `AuthorizationRepositoryService` instance can be specified for a domain. Currently, multiple `AuthorizationRepositoryService`'s cannot be used.

5.4.10. EventHandlingService SPI

`EventHandlingService` SPI is an interface that captures and processes various security events.

Using the `EventHandlingService` SPI, users can easily implement operations, including logging and sending a notification email to the administrator when a security event occurs, setting a lock on a subject, etc.

The basic concepts are very simple: the occurrences of `jeus.security.base.Event`'s in the security service will be notified to all `EventHandlingService`'s in the domain. Then `EventHandlingService` handles the event according to its content.

For example, an `EventHandlingService` can be used when a lock needs to be applied to a Subject. Whenever the `AuthenticationService` fails to authenticate a Subject, it will generate a `security.authentication.failed` Event. The lockout `EventHandlingService` catches this event and executes the `handleEvent(Event)` method. In the method, a login counter is kept and when it reaches a certain value (such as 3), a lock credential is applied to the Subject.

Then `SubjectValidationService` confirms that the subject is locked, and as a result, all attempts to log in using this Subject will fail. Thus, by using `EventHandlingService` with `SubjectValidationService`, user can implement a function that prohibits a subject from logging into the system if it fails to log in three times in a row.

However, the most common use of the `EventHandlingService` SPI is probably to implement various kinds of security loggers that record events. The events can be written in a text file, an XML file, or sometimes in an encrypted file to prevent repudiation.

Zero or more `EventHandlingService`'s can be configured for each security domain.

5.4.11. Dependencies between SPI Implementations

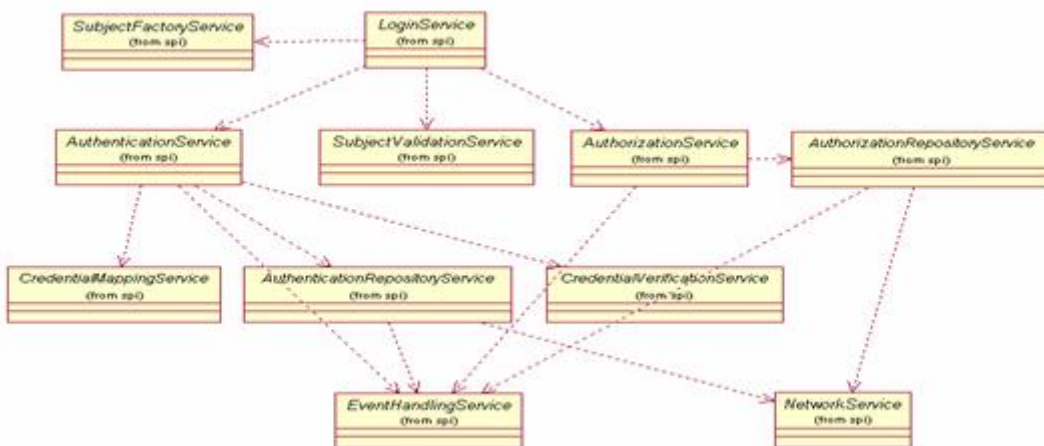
There may be a dependency between different SPI classes. For example, the `AuthenticationService` class depends on the `AuthenticationRepositoryService` class.



Here, dependency is simply the fact that one SPI implementation class calls a static method of another SPI implementation class. If the implementation class of `AuthenticationService` is `AuthenticationServiceImpl`, the class can obtain the information about the subject by calling the `AuthenticationRepositoryService.getSubject()` method.

Note that the fact that a dependency between SPI classes "may" exists implies that the dependency could exist but is not mandatory. It is dependent on the actual implementation of the SPI class. No SPI class directly calls the methods of another SPI class (with a few minor exceptions). In general, the dependency is generated when a SPI implementation class calls a method of a different SPI implementation class.

The following shows the dependencies between SPI implementations and SPI static methods in a default security system implementation.



Dependencies Between Default SPI Implementation Classes in the Default Security System

5.5. Security Services Configurations

After a new SPI implementation class is compiled, the class must be registered with the JEUS security system. For more information, refer to [Configuring Security Domain Components](#).

6. Using JACC Provider

This chapter describes the purpose of JACC and how to implement Custom JACC Provider to use it in the JEUS security system.

6.1. Overview

Jakarta Authorization was first introduced in J2EE version 1.4 with the name of Java Authorization Contract for Containers (JACC), and the current version is 2.0.

JACC serves two basic purposes.

- Provide a standardized SPI for EJB and Servlet authorization.
- Ensure compatibility between existing Java SE and Jakarta EE security models.

In short, JACC represents a standardized way of implementing an authorization provider that defines and handles authorization for EJBs and Servlets. Thus, if a JACC provider is implemented, it can be used in any JACC-compatible Jakarta EE server.

6.2. Introducing JACC Protocol

The entire JACC specification is divided into three parts as follows:

- Provider Protocol
- Policy Protocol
- Policy decision and execution protocol



For more information about JACC specifications (JSR-115), refer to JACC 2,0 specification in [Jakarta EE Authorization 2.0 Specification](#).

6.2.1. Provider Configuration Protocol

The JACC Provider configuration protocol describes how a JACC provider (JACC implementation) is announced and incorporated into the runtime environment of an application server. This protocol defines how JACC providers are registered with Jakarta EE servers. In general, this process is very simple as follows:

1. The Custom `java.security.Policy` class name is configured in the `jakarta.security.jacc.policy.provider` system properties. The JACC provider contains the custom `java.security.Policy` class.
2. The Jakarta EE server reads this class name and creates a class instance, and then casts it to the `java.security.Policy` type.

3. If successful, the default Java SE policy class will be replaced with the new JACC policy class by calling the `java.security.Policy.setPolicy()` method.
4. After the completion of the previous steps, all authorization checks will be performed by the new JACC policy. The current JACC policy object can be obtained using the `java.security.Policy.getPolicy()` method.



For more information about the JACC Provider configuration protocol, refer to JACC specifications.

6.2.2. Policy Configuration Protocol

The Policy Configuration protocol essentially defines how the security constraints defined in Jakarta EE deployment descriptor file (`ejb-jar.xml`, `web.xml`) are to be mapped to the `java.security.Permission` set defined in the `jakarta.security.jacc` package. It also defines how these Permissions are to be added to the JACC provider (Custom `java.security.Policy`).

This helps the provider to make correct authorization decisions for EJB and Servlet modules.



The Policy Configuration protocol does NOT define how to configure principal-to-role mappings. It only describes how to configure role-to-resource mappings based on the information in the standard Jakarta EE DD file. The principal-to-role mappings are defined according to the JACC provider vendor.

Policy can be configured as follows:

1. Specify the fully qualified class name of Custom `jakarta.security.jacc.PolicyConfigurationFactory` in the `jakarta.security.jacc.PolicyConfigurationFactory.provider` system property ("-D" property).

The JACC provider contains Custom `jakarta.security.jacc.PolicyConfigurationFactory` class.

2. The Jakarta EE server reads the property and creates a class instance (PCF).
3. Deploy the servlet and EJB modules.
4. The deployment code parses the provided `web.xml` and `ejb-jar.xml` DD files and converts the security constraints to JACC permission instances. The permission class is included in the `jakarta.security.jacc` package. Each permission instance represents the Role-to-Resource mapping defined in the servlet and EJB.
5. The deployment code calls the `PCF.getPolicyConfiguration()` method and receives an instance of the `jakarta.security.jacc.PolicyConfiguration` type.
6. The deployment code adds the role-to-resource permissions created in step 4 to `PolicyConfiguration` using various `PolicyConfiguration` methods.
7. After all permissions have been added, the `PolicyConfiguration.commit()` method is called.

When a Servlet and/or EJB module is undeployed, `PolicyConfiguration.delete()` method is called. This

removes all the permissions of the Servlet/EJB module.



For more information about the policy configuration process, refer to the JACC specifications.

6.2.3. Policy Decision and Execution Protocol

The policy decision and execution protocol describe how (EJB and Servlet) authorization decisions are to be carried out at runtime. They are applied after the requirements of the previous two protocols have been met.

Essentially, Policy decision and execution work as follows. This section uses a servlet as an example, but it also applies equally to an EJB.

1. An HTTP request is sent to the Servlet page.
2. The Servlet container configures some JACC context information using the `jakarta.security.jacc.PolicyContext` class.
3. The Servlet container then constructs two JACC Web permission instances (defined in the `jakarta.security.jacc` package). These permission instances represent the permission defined for the current Servlet page.
4. The Servlet container queries the JACC policy provider to see if the servlet requester has the two permissions mentioned in step 3. There are several ways to interpret the query.

For example, the `Policy.implies()` method can be used to interpret the query. In this case, all permissions, which are checked by the `java.security.ProtectionDomain` that is initialized by the principal of the requester, are used as parameters.

5. The JACC provider receives the request and determines if the requester has the permission to access the servlet page by using the following information: context information set from step 2, the permission instances created in step 3, the current principal(s), the principal-to-role mapping, and the role-to-resource mapping.
6. If the outcome of the authorization check is positive, the Servlet container will allow access to the Servlet page. Otherwise, an authorization error page will be returned.



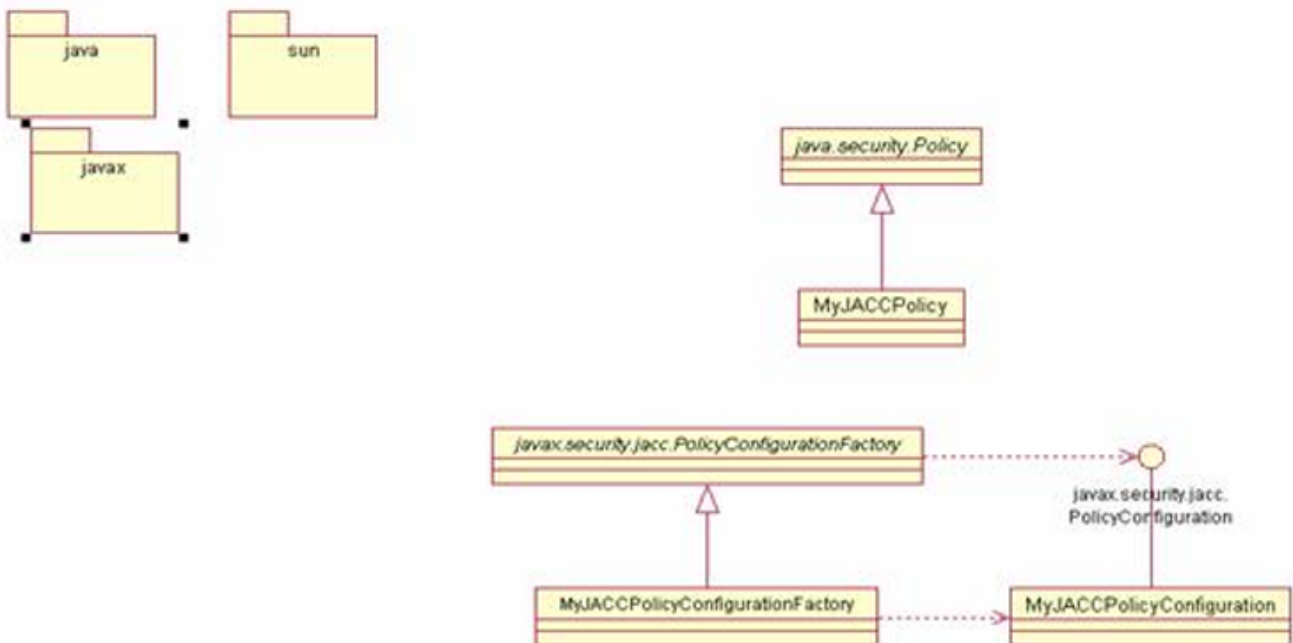
For more information about policy decision and execution protocol, refer to the JACC specifications.

6.3. Developing JACC Provider

This section describes how to develop the Custom JACC provider and some development instructions.

6.3.1. Implementing JACC Provider

The following is a simple picture that shows the relationships between the classes that need to be implemented for your JACC provider. In the figure, the classes that need to be implemented are labeled using names starting with “MyJACC”. Any name can be chosen for these classes.



JACC Provider Class

The following list shows the classes that are required for a complete implementation of a JACC provider:

- **java.security.Policy**

The heart of a JACC provider implementation is usually creating a subclass of the `java.security.Policy` class. The class is used in JACC for authorization purposes.

The usual way of implementing the subclass of `java.security.Policy` is by defining a new class, we call it `MyJACCPolicy` here, that extends `java.security.Policy`. Then, `MyJACCPolicy` overrides the `implies()` method in order to implement the policy decision and protocol that are described in [Policy Decision and Execution Protocol](#). When implementing the `implies()` method, both the permissions that are added using the `PolicyConfiguration` interface and the principal-to-role mappings must be considered in order to interpret an authorization query.

`MyJACCPolicy` must provide a public constructor without parameters, so that the Jakarta EE server can easily create a class instance.

- **jakarta.security.jacc.PolicyConfigurationFactory**

In order to satisfy the Policy configuration protocol, the application server must be able to add permission instances to JACC `java.security.Policy`. This is accomplished using the `jakarta.security.jacc.PolicyConfiguration` interface. The abstract class, `jakarta.security.jacc.PolicyConfigurationFactory`, is used to create the `PolicyConfiguration`

instance.

Thus, in order to create a complete JACC provider, you must implement the `getPolicyConfiguration()` method in the `jakarta.security.jacc.PolicyConfigurationFactory` class. This means that you must create a new subclass of the `jakarta.security.jacc.PolicyConfigurationFactory` class. Here, we call the new class `MyJACCPolicyConfigurationFactory`. `MyJACCPolicyConfigurationFactory` must be a concrete class with a public no-argument constructor, so that it can be easily instantiated by the application server.

The `MyJACCPolicyConfigurationFactory` must provide a public no-argument constructor to enable the Jakarta EE server to easily create instances of the class.

- **jakarta.security.jacc.PolicyConfiguration**

In order to implement the Policy configuration protocol, the application server must be able to add permission instances to the JACC `jakarta.security.Policy` implementation. This is accomplished through an instance of the `javax.security.jacc.PolicyConfiguration` interface. An instance of this interface is created through the abstract class, `jakarta.security.jacc.PolicyConfigurationFactory`, as already mentioned.

Implementing the `jakarta.security.jacc.PolicyConfiguration` interface is thus a requirement to complete a JACC provider. We will call our implementation class `MyJACCPolicyConfiguration`.

`jakarta.security.jacc.PolicyConfigurationFactory` implementation (`MyJACCPolicyConfigurationFactory`) should always return an instance of the `MyJACCPolicyConfiguration` class.



This section briefly explains how to implement each of the classes. For more information about how to implement the `javax.security.jacc.PolicyConfiguration` class, refer to the Policy Configuration Protocol of the JACC specification, and the Jakarta EE Javadoc for each class.

6.3.2. Packaging JACC Provider

In general, the JACC Provider and the supporting classes referenced by the JACC provider should be packaged together into a JAR file and be delivered, in a JAR format, to the target application server. We will call the JAR file, “`MyJACCPProvider.jar`”.

The path of the JAR file for the JACC provider must be included in the application server path. If needed, certain system properties can be configured, but the procedure for this varies slightly for different application servers. How to configure this for JEUS will be explained in [Integrating JACC Providers with the JEUS Security System](#).

6.3.3. Default JACC Provider

The JEUS security system provides a very simple default JACC provider. In general, it is strongly

discouraged to use this default provider as it was mainly developed for test purposes. Instead, the default standard authorization provider described in the earlier chapter is recommended for use.

However, if you wish to use another JACC provider, you must create your own JACC JAR archive and include the JACC provider file in it since there is no support for commercial products.

6.4. Integrating JACC Providers with the JEUS Security System

This section describes how to integrate JACC providers with the JEUS security system.

The process to integrate the JEUS security system with a JACC Provider is as follows.

1. Implement a Principal-to-Role Mapper

The JACC interface does not contain any implementation related to principal-to-role mapping, but only for role-to-resource mapping. The user must implement a separate JEUS-specific interface to implement the mapping, and JEUS provides the `isjeus.security.impl.aznrep.JACCPrincipalRoleMapper` interface for this purpose. This interface contains a single method, that needs to be implemented, called `addPrincipalRoleMapping(PermissionMap map, String policyId)`. This method adds the principal-to-role mapping to `PolicyConfiguration`, which is represented by the `policyId`.



Note that principal-to-role mappings have application scope in Jakarta EE. This is because all the principal-to-role mappings in an application are merged into a single map. Refer to the API documentation for more information about the `PermissionMap` class and about the `add()` method, that is used in merging `PermissionMap` instances.

The implementation of the `JACCPrincipalRoleMapper` interface must provide a public no-argument constructor, and it must be added to the JAR file of the JACC Provider. For more information on this, refer to [References](#) and Javadoc.

The process of the `JACCPrincipalRoleMapper` interface creating the principal-to-role mapping is as follows:

- a. The class name that implements the `jeus.security.jacc.principalRoleMapper` class is configured to the system property, `jeus.security.jacc.principalRoleMapper`.
- b. The class that implements the `jeus.security.impl.aznrep.JACCAuthorizationRepositoryService` class reads this property to create an instance by calling the `Class.forName(mapperClassname).newInstance()` method.
- c. Then the `addPrincipalRoleMapping()` method of the instance is called to create and add the principal-to-role mappings, that are defined in the JEUS DD file.

2. Setting the Security Configuration File

The JEUS security system uses two adapter classes to connect JEUS native authorization API with the JACC authorization API.

The roles of the two adapter classes are as follows:

- `jeus.security.impl.azn.JACCAuthorizationService`

Implements the container part of the Policy Decision and Execution protocol by invoking the `java.security.Policy.getPolicy().implies()` method to check for authorization.

- `jeus.security.impl.aznrep.JACCAuthorizationRepositoryService`

Implements the Policy Configuration protocol by allowing the container-generated `jeus.security.base.Policy` instance to be added to the `PolicyConfiguration` instance, which is a configuration component of the JACC provider.

In order to enable JACC, both of these security Services must be configured in the domain service definition of the `security-domains.xml` file.

JACC Security Configuration File: `<security-domains.xml>`

```
<?xml version="1.0"?>
<security-domains>
  ...
  <security-domain>
    <name>JACC_DOMAIN</name>
    <authorization>
      <jacc-service/>
    </authorization>
  </security-domain>
  . . .
</security-domains>
```

3. Adding the JACC Provider JAR file to the System Path

To add the JACC provider JAR file to the system path, simply place the JAR file in the following directory.

```
JEUS_HOME/lib/system
```

4. Setting Java System Properties

The JACC protocol and JEUS specifies three Java system properties to enable the application server to detect the presence of the JACC provider. These properties are as follows:

- `jakarta.security.jacc.policy.provider`

The name of the class that represents the JACC Provider, and implements `java.security.Policy`.

- `jakarta.security.jacc.PolicyConfigurationFactory.provider`

The name of the class that implements PolicyConfigurationFactory that creates and loads PolicyConfiguration instances.

- jeus.security.jacc.principalRoleMapper

The name of the class that implements the jeus.security.impl.aznrep.JACCPPrincipalRoleMapper interface, and creates the principal-to-role mapping from JEUS DD file.

The previous three system properties must be configured in the <jvm-option> element of domain.xml.

Java System Property Configuration for JACC <domain.xml>

```
<?xml version="1.0"?>
<domain xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <servers>
    <server>
      <name>server1</name>
      <!-- server JVM option -->

      <jvm-config>
        . . .
        <jvm-option>
          -Djakarta.security.jacc.policy.provider=
            myprovider.MyJACCPolicy
        </jvm-option>
        <jvm-option>
          -Djakarta.security.jacc.PolicyConfigurationFactory.provider=
            myprovider.MyJACCPolicyConfigurationFactory
        </jvm-option>
        <jvm-option>
          -Djeus.security.jacc.principalRoleMapper=
            myprovider.MyJACCPPrincipalToRoleMapper
        </jvm-option>
      </jvm-config>
    </server>
  </servers>
</domain>
```

Default JACC Provider Class Name

As already mentioned, the default class names for JACC provider are as follows:

Classification	Class Name
Policy	jeus.security.impl.jacc.JACCPolicyWrapper
PolicyConfigurationFactory	jeus.security.impl.jacc.JACCPolicyConfigurationFactoryImpl
JACCPPrincipalRoleMapper	jeus.security.impl.jacc.JACCDefaultPrincipalRoleMapper

These classes are packaged and placed in the following location.

JEUS_HOME/lib/system/jeus.jar

7. Using JAAS

This chapter introduces Java Authentication and Authorization Service (JAAS) and how to integrate SunOne Directory Server with the JEUS security system.

7.1. Overview

JAAS stands for the Java Authentication and Authorization Service. It implements the Java version of standard Pluggable Authentication Module (PAM), and authenticates and performs access control by supporting user based authentication.

In Java 2 SDK, and Standard Edition (J2SDK) 3, JAAS (Java™ Authentication and Authorization Service) was an optional package (extension). It has been integrated since the J2SDK 4.

The main purposes of JAAS are as follows.

- **Authenticates** users.

Uses secure method to authenticate the user who is executing the Java code, regardless of the type of code (application, applet, bean, or servlet).

- **Approves** users.

Confirms that the user maintains the access control required to execute the action.

Java 2 has been providing the access control (based on the source of the code and the signature) of the code source base. However, since the additional function of performing the access control based on the executor is not enough, JAAS supports the extended Java2 security architecture.

JAAS authentication is executed using the **plug-in availability** method. Thus, an application works separately from the authentication-based technology. A new or updated authentication technology can be used as a plug-in within the application, which eliminates the need to modify the application.

The application validates the authentication process by instantiating the LoginContext object. The LoginContext object decides on the authentication technology or LoginModule for use based on the configurations. The basic LoginModule verifies a user with the entered username and password, and some modules can verify a user through a voice recognition, fingerprints, or object.

If the user is authenticated, the approval component of JAAS protects the access to the resources by working with the core Java access control model. In J2SDK 3, access control was determined only by the location of the code and CodeSource. However, in J2SDK 4, it is determined by the source of the execution code and the user or service which can be represented by a Subject object that executes the code. If the authentication is successful, the LoginModule updates the subject by using the relevant principal and qualifications.



For more information about JAAS specification, refer to [JAAS Tutorial](#).

For basic integration between LDAP, to which the JAAS mechanism has been applied, a database, and the JEUS security system, you must extend and implement the LoginModule. The JEUS security system provides the services related to login and approval.

The following sections show examples to explain how LDAP and database work together by applying core JAAS related classes and interfaces to the JEUS security system.

- Example of integrating LDAP to LoginModule
- Example of integrating DBRealm to LoginModule

7.2. Implementing LoginModule to Integrate JEUS with LDAP

The `javax.security.auth.login.LoginContext` interface class provides the basic method used for authentication. This class enables a user to develop an application, which provides a particular authentication type, without depending on the login authentication technology.

As shown in the following, the JAAS authentication mechanism, which is extended from the LoginModule, is provided to work with LDAP. The authentication mechanism is performed through the `LdapLoginModule` that inherits the LoginModule interface.

`LdapLoginModule` has been extended to enable integration with the JEUS security system.

`jeus.security.impl.login.LdapLoginModule`

```
package jeus.security.impl.login;

import jeus.security.base.Domain;
import jeus.security.resource.Password;
import jeus.security.resource.PrincipalImpl;
import jeus.security.resource.RolePrincipalImpl;
import jeus.util.logging.JeusLogger;

import javax.security.auth.Subject;
import javax.security.auth.callback.*;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
import java.security.Principal;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;

public class LdapLoginModule implements LoginModule {
    protected static final JeusLogger logger = (JeusLogger)
JeusLogger.getLogger("jeus.security.loginmodule");

    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;

    private boolean succeeded = false;
```

```

private boolean commitSucceeded = false;

private String username;
private String password;
private String domain;

private Principal userPrincipal;
private Password userCredential;

private SunOneLdapAuthenticator authenticator;

private static final String INITIAL_CONTEXT_FACTORY = "initialContextFactory"; // optional
private static final String PROVIDER_URL = "providerURL";
private static final String CONNECTION_USERNAME = "connectionUsername";
private static final String CONNECTION_PASSWORD = "connectionPassword";
private static final String USER_BASE = "userBase";
private static final String USER_SEARCH_MAPPING = "userSearchMapping";
private static final String USER_PASSWORD_ATTR = "userPasswordAttr";
private static final String USER_ROLE_ATTR = "userRoleAttr";
private static final String ROLE_BASE = "roleBase";
private static final String ROLE_NAME_ATTR = "roleNameAttr";
private static final String ROLE_SEARCH_MAPPING = "roleSearchMapping";

private final String SUN_JDK_LDAP_CONTEXT_FACTORY = "com.sun.jndi.ldap.LdapCtxFactory";

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

    this.subject = subject;
    this.callbackHandler = callbackHandler;
    this.domain = Domain.SYSTEM_DOMAIN_NAME;

    authenticator = new SunOneLdapAuthenticator();
    initAuthenticator(authenticator, options);
}

private void initAuthenticator(SunOneLdapAuthenticator authenticator, Map options) {
    // INITIAL_CONTEXT_FACTORY
    String value = (String) options.get(INITIAL_CONTEXT_FACTORY);
    if (value == null) {
        authenticator.setContextFactory(SUN_JDK_LDAP_CONTEXT_FACTORY);
    } else {
        authenticator.setContextFactory(value);
    }

    authenticator.setProviderUrl((String) options.get(PROVIDER_URL));
    authenticator.setConnectionUsername(
        (String) options.get(CONNECTION_USERNAME));
    authenticator.setConnectionPassword(
        (String) options.get(CONNECTION_PASSWORD));

    value = (String) options.get(USER_BASE);
    if (value != null) {
        authenticator.setUserBase(value);
    } else {
        String msg = "LoginMoulde initialization failed. " + "userBase option missing.";
        logger.warning(msg);
        throw new IllegalArgumentException(msg);
    }
}

```



```

    authenticator.setUserPasswordAttr((String) options.get(USER_PASSWORD_ATTR));

    value = (String) options.get(USER_SEARCH_MAPPING);
    if (value != null) {
        authenticator.setUserSearchMapping(new MessageFormat(value));
    } else {
        String msg = "LoginMoulde initialization failed. " + "userSearchMapping option missing.";
        logger.warning(msg);
        throw new IllegalArgumentException(msg);
    }

    authenticator.setUserRoleAttr((String) options.get(USER_ROLE_ATTR));

    value = (String) options.get(ROLE_BASE);
    if (value != null) {
        authenticator.setRoleBase(value);
    } else {
        String msg = "LoginMoulde initialization failed. " + "roleBase option missing.";
        logger.warning(msg);
        throw new IllegalArgumentException(msg);
    }

    value = (String) options.get(ROLE_NAME_ATTR);
    if (value != null) {
        authenticator.setRoleNameAttr(value);
    } else {
        String msg = "LoginMoulde initialization failed. " + "roleNameAttr option missing.";
        logger.warning(msg);
        throw new IllegalArgumentException(msg);
    }

    value = (String) options.get(ROLE_SEARCH_MAPPING);
    if (value != null) {
        authenticator.setRoleSearchMapping(new MessageFormat(value));
    } else {
        String msg = "LoginMoulde initialization failed. " + "roleSearchMapping option missing.";
        logger.warning(msg);
        throw new IllegalArgumentException(msg);
    }
}

public boolean commit() throws LoginException {
    if (succeeded == false) {
        return false;
    } else {
        userPrincipal = new PrincipalImpl(username);
        if (!subject.getPrincipals().contains(userPrincipal))
            subject.getPrincipals().add(userPrincipal);

        ArrayList roles = authenticator.getRoles();
        for (Iterator i = roles.iterator(); i.hasNext(); ) {
            String roleName = (String) i.next();
            logger.fine("Adding role to subject : username = " + username + ", roleName = " +
roleName);
            subject.getPrincipals().add(new RolePrincipalImpl(roleName));
        }

        userCredential = new Password(password);
    }
}

```

```

        subject.getPrivateCredentials().add(userCredential);

        username = null;
        password = null;
        domain = null;
        commitSucceeded = true;
        return true;
    }
}

public boolean abort() throws LoginException {
    if (succeeded == false) {
        return false;
    } else if (succeeded == true && commitSucceeded == false) {
        succeeded = false;
        username = null;
        password = null;
        domain = null;
        userPrincipal = null;
        userCredential = null;
    } else {
        logout();
    }
    return true;
}

public boolean logout() throws LoginException {
    subject.getPrincipals().remove(userPrincipal);
    subject.getPrivateCredentials().remove(userCredential);
    succeeded = false;
    succeeded = commitSucceeded;
    username = null;
    password = null;
    domain = null;
    userPrincipal = null;
    userCredential = null;
    return true;
}

public boolean login() throws LoginException {
    Callback[] callbacks = null;

    // prompt for a user name and password
    if (callbackHandler == null)
        throw new LoginException("Error: no CallbackHandler available " +
                                   "to garner authentication information from the user");

    callbacks = new Callback[3];
    callbacks[0] = new NameCallback("user name: ");
    callbacks[1] = new PasswordCallback("password: ", false);
    callbacks[2] = new TextInputCallback("domain: ");
    try {
        callbackHandler.handle(callbacks);
        username = ((NameCallback) callbacks[0]).getName();
        char[] tmpPassword = ((PasswordCallback) callbacks[1]).getPassword();
        if (tmpPassword == null) {
            tmpPassword = new char[0];
        }
        password = new String(tmpPassword);
    }
}

```

```

        ((PasswordCallback) callbacks[1]).clearPassword();
        domain = ((TextInputCallback) callbacks[2]).getText();

        if (!authenticator.authenticate(username, password)) {
            throw new LoginException("LDAP authentication failed.");
        }
        succeeded = true;
    } catch (UnsupportedCallbackException uce) {
        uce.printStackTrace();
        LoginException le = new LoginException(
            "Error: " + uce.getCallback().toString() +
            " not available to garner authentication information " +
            "from the user");
        le.initCause(uce);
        throw le;
    } catch (Exception e) {
        e.printStackTrace();    // todo. logging
        if (e instanceof LoginException) {
            throw (LoginException) e;
        } else {
            LoginException le = new LoginException(e.toString());
            le.initCause(e);
            throw le;
        }
    }
    return succeeded;
}
}

```



The RolePrincipalImpl information needs to be converted to the RolePrincipalImpl type, because the user and role information, which is added during runtime, must be applied to the JEUS security system.

7.3. Configuring LDAP JAAS LoginModule Service

To provide the login service by applying the JAAS LoginModule to a particular domain in the JEUS security system, specify the <jaas-login-config> element when configuring the domain security service. For more information, refer to the '**authentication**' section in [Configuring Security Service](#).

To provide the LDAP LoginModule security service to DEFAULT_APPLICATION_DOMAIN, configure the login service and approval service as follows.

Configuring LDAP JAAS LoginModule Service: <security-domains.xml>

```

<?xml version="1.0" encoding="UTF-8"?>
<security-domains xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9.0">
    . . .
    <security-domain>
        <name>DEFAULT_APPLICATION_DOMAIN</name>
        <authentication>
            <jaas-login-config>
                <login-module>

```

```

    <login-module-classname>
        jeus.security.impl.login.LdapLoginModule
    </login-module-classname>
    <control-flag>required</control-flag>
    <option>
        <name>initialContextFactory</name>
        <value>com.sun.jndi.ldap.LdapCtxFactory</value>
    </option>
    <option>
        <name>providerURL</name>
        <value>ldap://192.168.1.63:389</value>
    </option>
    <option>
        <name>connectionUsername</name>
        <value>cn=Directory Manager</value>
    </option>
    <option>
        <name>connectionPassword</name>
        <value>adminadmin</value>
    </option>
    <option>
        <name>userBase</name>
        <value>ou=People,dc=sample,dc=com</value>
    </option>
    <option>
        <name>userSearchMapping</name>
        <value>(uid={0})</value>
    </option>
    <option>
        <name>roleBase</name>
        <value>ou=Groups,dc=sample,dc=com</value>
    </option>
    <option>
        <name>roleNameAttr</name>
        <value>cn</value>
    </option>
    <option>
        <name>roleSearchMapping</name>
        <value>(uniqueMember={0})</value>
    </option>
</login-module>
</jaas-login-config>
</authentication>
<authorization>
    <repository-service>
        <custom-repository>
            <classname>
jeus.security.impl.aznrep.CustomPolicyFileRealmAuthorizationRepositoryService
            </classname>
            <property>
                <name>PolicyClassName</name>
                <value>jeus.security.base.CustomJeusPolicy</value>
            </property>
            <property>
                <name>UserPrincipalClassName</name>
                <value>jeus.security.resource.PrincipalImpl</value>
            </property>
            <property>
                <name>RolePrincipalClassName</name>

```

```

        <value>jeus.security.resource.RolePrincipalImpl</value>
    </property>
</custom-repository>
</repository-service>
</authorization>
<security-domain>
. . .
</security-domains>

```

The following is the description for each class.

- jeus.security.impl.callback.JAASUsernamePasswordCallbackHandler

Provides the basic mechanism for obtaining authentication information (username and password) for LoginModule of the JEUS security system.

- jeus.security.impl.login.LdapLoginModule

Supports LoginModule based on the LDAP Attribute value that is defined as an option value in the JEUS security system.

- jeus.security.impl.aznrep.CustomPolicyFileRealmAuthorizationRepositoryService

Provides the authorization service by applying the UserPrincipalClassName/RolePrincipalClassName type, defined by the user, in the JEUS security system.

- jeus.security.base.CustomJeusPolicy

Extends and implements the jeus.security.base.Policy class. This class supports principal-to-role mapping even when there is no jeus-web-dd.xml descriptor at runtime in the JEUS security system and when the LoginModule defines the RolePrincipalImpl class for the Principal.

When you are finished with the configurations, start JEUS, and then start the login service for the application deployed to DEFAULT_APPLICATION_DOMAIN.

7.4. Implementing LoginModule to Integrate with Database

The authentication mechanism is performed through the DBRealmLoginModule that inherits the LoginModule interface. To integrate with the JEUS security system, extend and implement the DBRealmLoginModule as shown in the following.

jeus.security.impl.login.DBRealmLoginModule

```

package jeus.security.impl.login;

import jeus.security.base.Domain;
import jeus.security.base.ServiceException;
import jeus.security.resource.Password;

```

```

import jeus.security.resource.PrincipalImpl;
import jeus.security.resource.RolePrincipalImpl;
import jeus.util.logging.JeusLogger;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.*;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
import javax.sql.DataSource;
import java.security.Principal;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

/**
 * User: choco
 * Date: 2016. 09. 13
 * Time: PM 4:35:57
 */
public class DBRealmLoginModule implements LoginModule {
    protected static final JeusLogger logger = (JeusLogger)
JeusLogger.getLogger("jeus.security.login");
    protected String dsExportName;
    protected String principalsQuery = "select password from jeus_users where username=?";
    protected String rolesQuery = "select role from jeus_roles where username=?";

    private String username;
    private String password;
    private String domain;

    private Subject subject;
    private CallbackHandler callbackHandler;
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    protected Map options;
    private Principal userPrincipal;
    private Password userCredential;

    public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map
options) {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.options = options;

        try {
            domain = Domain.getCurrentDomain().getName();
        } catch (ServiceException e) {
            domain = Domain.SYSTEM_DOMAIN_NAME;
        }
    }

```

```

dsExportName = (String) options.get("exportName");
if (dsExportName == null) {
    String msg = "LoginMoulde initialization failed. " + "exportName option missing.";
    logger.warning(msg);
    throw new IllegalArgumentException(msg);
}

Object tmp = options.get("principalsQuery");
if (tmp != null)
    principalsQuery = tmp.toString();
tmp = options.get("rolesQuery");
if (tmp != null)
    rolesQuery = tmp.toString();
logger.debug("DBRealmLoginModule, export name : " + dsExportName);
logger.debug("principalsQuery=" + principalsQuery);
logger.debug("rolesQuery=" + rolesQuery);
logger.debug("initialize successfully");
}

public boolean login() throws LoginException {
    Callback[] callbacks = null;

    // prompt for a user name and password
    if (callbackHandler == null)
        throw new LoginException("Error: no CallbackHandler available " + "to garner
authentication information from the user");

    callbacks = new Callback[3];
    callbacks[0] = new NameCallback("user name: ");
    callbacks[1] = new PasswordCallback("password: ", false);
    callbacks[2] = new TextInputCallback("domain: ");
    try {
        callbackHandler.handle(callbacks);
        username = ((NameCallback) callbacks[0]).getName();
        char[] tmpPassword = ((PasswordCallback) callbacks[1]).getPassword();
        if (tmpPassword == null) {
            tmpPassword = new char[0];
        }
        password = new String(tmpPassword);
        ((PasswordCallback) callbacks[1]).clearPassword();
        domain = ((TextInputCallback) callbacks[2]).getText();

        String expectedPassword = getUsersPassword();
        if (validatePassword(password, expectedPassword) == false) {
            throw new LoginException("DBRealm authentication failed.");
        }
        succeeded = true;
    } catch (UnsupportedCallbackException uce) {
        uce.printStackTrace();
        LoginException le = new LoginException(
            "Error: " + uce.getCallback().toString() +
            " not available to garner authentication information " +
            "from the user");
        le.initCause(uce);
        throw le;
    } catch (Exception e) {
        e.printStackTrace();    // todo. logging
        if (e instanceof LoginException) {
            throw (LoginException) e;
        }
    }
}

```

```

        } else {
            LoginException le = new LoginException(e.toString());
            le.initCause(e);
            throw le;
        }
    }
    return succeeded;
}

private String getUsersPassword() throws LoginException {
    String password = null;
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;

    try {
        InitialContext ctx = new InitialContext();
        DataSource ds = (DataSource) ctx.lookup(dsExportName);
        conn = ds.getConnection();
        ps = conn.prepareStatement(principalsQuery);
        ps.setString(1, username);
        rs = ps.executeQuery();
        if (rs.next() == false)
            throw new FailedLoginException("No matching username found");

        password = rs.getString(1);
    }
    catch (NamingException ex) {
        throw new LoginException(ex.toString(true));
    }
    catch (SQLException ex) {
        logger.debug("Query failed", ex);
        throw new LoginException(ex.toString());
    }
    finally {
        if (rs != null) {
            try {
                rs.close();
            }
            catch (SQLException e) {
            }
        }
        if (ps != null) {
            try {
                ps.close();
            }
            catch (SQLException e) {
            }
        }
        if (conn != null) {
            try {
                conn.close();
            }
            catch (SQLException ex) {
            }
        }
    }
    return password;
}

```



```

public boolean logout() throws LoginException {
    subject.getPrincipals().remove(userPrincipal);
    subject.getPrivateCredentials().remove(userCredential);
    succeeded = false;
    succeeded = commitSucceeded;
    username = null;
    password = null;
    domain = null;
    // trusted = false;
    userPrincipal = null;
    userCredential = null;
    return true;
}

public boolean commit() throws LoginException {
    if (succeeded == false) {
        return false;
    } else {
        userPrincipal = new PrincipalImpl(username);
        if (!subject.getPrincipals().contains(userPrincipal))
            subject.getPrincipals().add(userPrincipal);

        ArrayList roles = getRoleSets();
        for (Iterator i = roles.iterator(); i.hasNext();) {
            String roleName = (String) i.next();
            logger.debug("Adding role to subject : username = " + username + ", roleName = " +
roleName);
            subject.getPrincipals().add(new RolePrincipalImpl(roleName));
        }

        userCredential = new Password(password);
        subject.getPrivateCredentials().add(userCredential);

        username = null;
        password = null;
        domain = null;
        commitSucceeded = true;
        return true;
    }
}

public boolean abort() throws LoginException {
    if (succeeded == false) {
        return false;
    } else if (succeeded == true && commitSucceeded == false) {
        succeeded = false;
        username = null;
        password = null;
        domain = null;
        userPrincipal = null;
        userCredential = null;
    } else {
        logout();
    }
    return true;
}

protected ArrayList getRoleSets() throws LoginException {

```

```

Connection conn = null;
HashMap setsMap = new HashMap();
PreparedStatement ps = null;
ResultSet rs = null;
ArrayList roles = new ArrayList();
try {
    InitialContext ctx = new InitialContext();
    DataSource ds = (DataSource) ctx.lookup(dsExportName);
    conn = ds.getConnection();
    ps = conn.prepareStatement(rolesQuery);
    try {
        ps.setString(1, username);
    }
    catch (ArrayIndexOutOfBoundsException ignore) {
    }
    rs = ps.executeQuery();

    while (rs.next()) {
        String rolename = rs.getString(1);
        roles.add(rolename);
    }
}
catch (NamingException ex) {
    throw new LoginException(ex.toString(true));
}
catch (SQLException ex) {
    logger.debug("SQL failure", ex);
    throw new LoginException(ex.toString());
}
finally {
    if (rs != null) {
        try {
            rs.close();
        }
        catch (SQLException e) {
        }
    }
    if (ps != null) {
        try {
            ps.close();
        }
        catch (SQLException e) {
        }
    }
    if (conn != null) {
        try {
            conn.close();
        }
        catch (Exception ex) {
        }
    }
}

return roles;
}

protected boolean validatePassword(String inputPassword, String expectedPassword) {
    if (inputPassword == null || expectedPassword == null)
        return false;
}

```

```

    }
    return inputPassword.equals(expectedPassword);
}

```



The RolePrincipalImpl information needs to be converted to the RolePrincipalImpl type, because the user and role information, which is added during runtime, must be applied to the JEUS security system.

7.5. Configuring LoginModule Service

To provide the login service by applying the JAAS LoginModule to a particular domain in the JEUS security system, configure the <jaas-login-config> element for the domain security service. For the details, refer to **'authentication'** section in [Configuring Security Service](#).

To provide the Database LoginModule security service to DEFAULT_APPLICATION_DOMAIN, configure the login service and authentication service as follows:

Domain Service Configuration: <security-domains.xml>

```

<?xml version="1.0" encoding="UTF-8"?>
<security-domains xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
. . .
  <security-domain>
    <name>DEFAULT_APPLICATION_DOMAIN</name>
    <authentication>
      <jaas-login-config>
        <login-module>
          <login-module-classname>
            jeus.security.impl.login.DBRealmLoginModule
          </login-module-classname>
          <control-flag>required</control-flag>
          <option>
            <name>exportName</name>
            <value>dbrealmtest</value>
          </option>
          <option>
            <name>principalsQuery</name>
            <value>
              select password from DEFAULT_APPLICATION_DOMAIN_Principals
              where username=?
            </value>
          </option>
          <option>
            <name>rolesQuery</name>
            <value>
              select role from DEFAULT_APPLICATION_DOMAIN_roles
              where username=?
            </value>
          </option>
        </login-module>
      </jaas-login-config>
    </authentication>
  </security-domain>
</security-domains>

```

```

    <authorization>
      <repository-service>
        <custom-repository>
          <classname>
jeus.security.impl.aznrep.
            CustomPolicyFileRealmAuthorizationRepositoryService
          </classname>
          <property>
            <name>PolicyClassName</name>
            <value>jeus.security.base.CustomJeusPolicy</value>
          </property>
          <property>
            <name>UserPrincipalClassName</name>
            <value>jeus.security.resource.PrincipalImpl</value>
          </property>
          <property>
            <name>RolePrincipalClassName</name>
            <value>jeus.security.resource.RolePrincipalImpl</value>
          </property>
        </custom-repository>
      </repository-service>
    </authorization>
  <security-domain>
    . . .
  </security-domains>

```

The following is the description for each class.

- jeus.security.impl.login.DBRealmLoginModule

Supports LoginModule based on the exportname, principal, and role database query values. Optional for the JEUS security system.

The following are the list of options.

Item	Description
exportName	Sets the export-name of the database, defined in domain.xml.
principalsQuery	Defines a query, with one primary key, that obtains a password for the principal.
rolesQuery	Defines a query, with one primary key, that obtains roles of the principal.

- jeus.security.impl.aznrep.CustomPolicyFileRealmAuthorizationRepositoryService

Provides the authorization service by applying the UserPrincipalClassName/RolePrincipalClassName type, defined by the user, in the JEUS security system.

- jeus.security.base.CustomJeusPolicy

Extends and implements the jeus.security.base.Policy class. This class supports principal-to-role mapping even when there is no jeus-web-dd.xml descriptor at runtime in the JEUS security system and when the LoginModule defines the RolePrincipalImpl class for the Principal.



When you are finished with the configurations, start JEUS, and then start the login service for the application deployed to `DEFAULT_APPLICATION_DOMAIN`.

Appendix A: Security Event Service

This appendix describes the security event services.

A.1. Overview

This appendix describes the standard security events that are emitted to `EventHandlingService` from the SPI classes and default security service implementation classes. Use this reference to develop your own event handling providers by implementing the `jeus.security.spi.EventHandlingService` SPI.

The listing format is as follows:

```
G.2.X <Event type> = Event type
    Source Class: Class where the event occurred
    Event Type: Event type
    Event Level: Event level (FATAL, SERIOUS, WARNING, INFORMATION, DEBUG).
    Event Context: Key-value pairs for the event context.
    Emitted When? Conditions under which the event occurs
```

Normally, events are only emitted to an `EventHandlingService` that is in the same domain as the event source. This excludes two events, the `security.install.successful` and `security.uninstall.attempt` events, that are emitted to all configured domains in the security system.



For more information about the `jeus.security.base.Event` class and the `jeus.security.spi.EventHandlingService` class, refer to Javadoc.

A.2. Event

The following is the list of the standard security events.

security.validation.failed

Source Class	<code>jeus.security.spi.SubjectValidationService</code>
Event Type	<code>security.validation.failed</code>
Event Level	WARNING
Event Context	<ul style="list-style-type: none">◦ Key: "subject"◦ Value: <code>jeus.security.base.Subject</code> that failed validation.
Emitted When	Whenever a <code>SubjectValidationService</code> throws a <code>SecurityException</code>

security.authentication.failed

Source Class	jeus.security.spi.AuthenticationService
Event Type	security.authentication.failed
Event Level	WARNING
Event Context	<ul style="list-style-type: none">◦ Key: "subject"◦ Value: jeus.security.base.Subject that failed validation.
Emitted When	Whenever user authentication for the Subject fails.

security.authorization.failed

Source Class	jeus.security.spi.AuthorizationService
Event Type	security.authentication.failed
Event Level	WARNING
Event Context	<ul style="list-style-type: none">◦ Key: "contextid" Value: Context ID for which the permission was checked◦ Key: "permission" Value: java.security.Permission that needs to be checked.◦ Key: "subject" Value: jeus.security.base.Subject that failed user authentication.
Emitted When	Whenever the user authentication fails

security.authentication.repository.subject.added

Source Class	jeus.security.spi.AuthenticationRepositoryService
Event Type	security.authentication.repository.subject.added
Event Level	INFORMATION
Event Context	<ul style="list-style-type: none">◦ Key: "subject"◦ Value: jeus.security.base.Subject that is added
Emitted When	Whenever a Subject is successfully added to the AuthenticationRepositoryService.

security.authentication.repository.subject.removed

Source Class	jeus.security.spi.AuthenticationRepositoryService
---------------------	---

Event Type	security.authentication.repository.subject.removed
Event Level	INFORMATION
Event Context	<ul style="list-style-type: none"> ◦ Key: "subject" ◦ Value: jeus.security.base.Subject that is deleted.
Emitted When	Whenever a Subject is successfully removed from the AuthenticationRepositoryService.

security.authentication.repository.subject.removed.complete

Source Class	jeus.security.spi.AuthenticationRepositoryService
Event Type	security.authentication.repository.subject.removed.complete
Event Level	INFORMATION
Event Context	<ul style="list-style-type: none"> ◦ Key: "name" ◦ Value: The deleted Subject
Emitted When	Whenever a Subject is successfully removed from the AuthenticationRepositoryService.

security.authorization.repository.policy.added

Source Class	jeus.security.spi.AuthorizationRepositoryService
Event Type	security.authorization.repository.policy.added
Event Level	INFORMATION
Event Context	<ul style="list-style-type: none"> ◦ Key: "policy" ◦ Value: jeus.security.base.Policy that is added
Emitted When	Whenever a Policy is added to the AuthorizationRepositoryService.

security.authorization.repository.policy.removed

Source Class	jeus.security.spi.AuthorizationRepositoryService
Event Type	security.authorization.repository.policy.removed
Event Level	INFORMATION
Event Context	<ul style="list-style-type: none"> ◦ Key: "policy" ◦ Value: jeus.security.base.Policy that is deleted
Emitted When	Whenever Policy data is removed from the AuthorizationRepositoryService.

security.authorization.repository.policy.removed.complete

Source Class	jeus.security.spi.AuthorizationRepositoryService
Event Type	security.authorization.repository.policy.removed.complete
Event Level	INFORMATION
Event Context	<ul style="list-style-type: none">◦ Key: "contextid" Value: The java.lang.String type Context ID that was removed from the repository.
Emitted When	Whenever a context id is removed from the AuthorizationRepositoryService.

security.install.successful

Source Class	jeus.security.spi.SecurityInstaller
Event Type	security.install.successful
Event Level	INFORMATION
Event Context	None
Emitted When	After the security system has been successfully installed.

security.uninstall.attempt

Source Class	jeus.security.spi.SecurityInstaller
Event Type	security.uninstall.attempt
Event Level	INFORMATION
Event Context	None
Emitted When	Before the security system is to be uninstalled.

Appendix B: JEUS Server Permissions

This appendix describes the standard Permission resource names and resource actions.

B.1. Overview

This appendix lists the standard Permission resource names and resource actions. They are used by various JEUS sub-modules such as JNDI, JMS, manager, and security to check for authority to access the resources. The `java.security.Permission` type related to the resource Permission check is always `jeus.security.resource.ResourcePermission`, and the context id is always "default".

Authorization is always performed by combining the resource name with the resource action. In general, the resource name is the name of the target resource, and the resource action is the action that will be performed on the target. If the authorization configurations do not seem to be properly configured, check Master Server or the server logs and add the proper permissions.

B.2. JEUS System Resource Name

This section only describes the major resource names that are provided in the default JEUS security system.

Resource name	Description
<code>jeus.*</code>	Accesses all resource names in the JEUS system.
<code>jeus.server.<server-name>.*</code>	<p>Accesses all resource names of a particular server in the JEUS system. The list of permissions that are checked by the server when the default security system is used.</p> <p>Depending on the resource action, it is categorized as follows:</p> <ul style="list-style-type: none">◦ boot: When starting the server.◦ down: When terminating the server.◦ deploy: When deploying applications to the server.◦ ftp: When using ftp for file transfer.
<code>jeus.server.<server-name>.app.<application-name></code>	The resource name for a particular application of a specific server in the JEUS system.
<code>jeus.cluster.<cluster-name>.*</code>	Accesses all the resource names of a specific cluster of the JEUS system.

Resource name	Description
jeus.domain.<domain-name>	<p>The resource name for permission to dynamically modify the configurations in the JEUS system. The domain specified here is the JEUS system domain instead of the security domain.</p> <p>Depending on the resource action, it is categorized as follows:</p> <ul style="list-style-type: none"> ◦ dynamicConfiguration: When dynamically changing the domain configurations using jeusadmin.
jeus.jndi	<p>The resource name for JNDI operation permission of the JEUS system.</p> <p>Depending on the resource action, it is categorized as follows:</p> <ul style="list-style-type: none"> ◦ lookup: When trying to look up a object using JNDI. ◦ modify: When adding/deleting/changing a JNDI repository object such as bind/unbind/rename. ◦ list : When retrieving the list of objects that is stored in the JNDI repository.
jeus.node.<node-name>	<p>The resource name for a specific node in the JEUS system. Used when adding or deleting nodes. For more information about nodes, refer to "JEUS Server Guide".</p> <p>Depending on the resource action, it is categorized as follows:</p> <ul style="list-style-type: none"> ◦ edit: When adding or deleting a node.

B.3. jeusadmin Command Permission Configurations

Based on the default security system, the permissions can be set in units of jeusadmin command.

When permissions are set in units of jeusadmin command, only the permission for the command will be checked, and other internal permissions will be ignored.

The resource names of the command permissions are related to the command option. The resource names, explained earlier, are used for the server, servers, cluster, clusters, node options. For other options, the name, jeus.domain.<domain-name> is used as the resource name.

The resource action for the command permission is defined for the command name, and not for the command aliases. Thus, the actual command name, that can be checked using help <command-name> command, should be used. For more information about commands, refer to "Part II. Console Commands and Tools" in JEUS Reference Guide.

The following is an example that gives the administrator permission to user1, and the permission to perform deploy on server2 to user2. Since only the permission for server2 will be granted to user2, the resource name is set to "jeus.server.server2.*". The resource action is set to "deploy-application", which is the actual deployment command name in jeusadmin.

Security System Policy Configuration: <policies.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<policies xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <policy>
    <role-permissions>
      <role-permission>
        <principal>user1</principal>
        <role>adminRole</role>
      </role-permission>
      <role-permission>
        <principal>user2</principal>
        <role>server2DeployRole</role>
      </role-permission>
    </role-permissions>
    <resource-permissions>
      <context-id>default</context-id>
      <resource-permission>
        <role>adminRole</role>
        <resource>jeus.*</resource>
        <actions>*</actions>
      </resource-permission>
      <resource-permission>
        <role>server2DeployRole</role>
        <resource>jeus.server.server2.*</resource>
        <actions>deploy-application</actions>
      </resource-permission>
    </resource-permissions>
  </policy>
</policies>
```

References

- Java Authorization Contract for Containers Specification Version 1.0

Provides information about the JACC provider.

- Jakarta EE 9 Specification

Provides information on the basic security architecture applied to Jakarta EE servers including JEUS.

- EJB 4.0 Specification

Provides information about the EJB security model.

- Servlet 5.0 Specification

Provides information about the Servlet security model.

- Javadoc for java.security, javax.security.auth, jakarta.security.jacc

Provides detailed information about the basic classes related to security.

- Javadoc JEUS API

[JEUS_HOME/docs/api/jeusapi/index.html](#)

- XML Reference - domain.xml settings for security service

[JEUS_HOME/docs/reference/schema/index.html](#)