NCICB

Ekagra

High Impact – High Value – Business Results

Common Security Module

CSM Guide for Application Developers

Version No: 1.1

Last Modified: 3/21/05

Author : Vinay Kumar, Eric Copen, Kalpesh Patel, Kunal Modi

Team : Common Security Module (CSM)

Purchase Order# 34552

Client : National Cancer Institute - Center for Bioinformatics,
National Institutes of Health,
US Department of Health and Human Services

Document History

Document Location

The most current version of this document is located in CVS under security/docs.

Revision History

Version Number	*Revision Date*	*Author*	*Summary of Changes*
0.1	2/24/05	Vinay Kumar, Eric Copen, Kalpesh Patel	Initial Table of Contents
0.2	3/02/05	Eric Copen	Integrating separate existing documents into this document, adding Introduction and CSM Overview Text, and text in other places as needed.
0.3	3/04/05	Kunal Modi	Incorporated Comments and Document Restructuring
1.0	3/04/05	Eric Copen	Prepared for release
1.1	3/21/05	Eric Copen	Incorporated changes from technical writers

Review

Name	*Team/Role*	*Version*	*Date Reviewed*	*Reviewer Comments*
Vinay Kumar	Team Lead	0.2	2/14	Approved
JJ Maurer	Ekagra Management	0.2	2/15	Approved with minor changes
Jill Hadfield, Liz Lucchesi	Technical Writers	1.0	3/04 – 3/18	Made minor changes for caCORE Technical Guide

Related Documents

More information can be found in the following related CSM documents:

Document Name
UPT User Guide
Software Architecture Document
CSM Enterprise Architect Model
CSM Reference Implementation Guide

These and other documents can be found on the CSM website: http://ncicb.nci.nih.gov/core/CSM

Table of Contents

CSM Guide for Application Developers

1. Introduction to CSM

1.1 Purpose

This document provides all the information application developers need to successfully integrate with NCICB's Common Security Module (CSM). The CSM was chartered to provide a comprehensive solution to common security objectives so not all development teams would have to create their own security methodology. CSM is flexible enough to allow application developers to integrate security with minimal coding effort. This phase of the Common Security Module brings the NCICB team one step closer to the goal of application security management, single sign-on, and Health Insurance Portability and Accountability Act (HIPPA) compliance.

1.2 Scope

This document shows how to deploy and integrate the CSM services, including Authentication, Authorization, and User Provisioning. For specific questions regarding using the UPT, refer to the User Provisioning Tool (UPT) User Guide (http://ncicb.nci.nih.gov/core/CSM).

1.3 Using This Guide

You should begin by reading the *CSM Overview* on this page to learn the CSM concepts and how they apply to your own application. Next, *The Three Services* section on page 7 explains the three manager interfaces and the methods to incorporate them. The *Deployment Models* section on page 9 explains how to deploy the services and how to integrate with them. The deployment and integration sections (*Integrating with the CSS Authentication Service* on page 29, *Integrating with the CSM Authorization Service* on page 31 and *Integrating with the User Provisioning Service* on page 32) consist of multiple step-by-step guides to help you with a variety of configurations.

2. CSM Overview

2.1 Explanation

The CSM provides application developers with powerful security tools in a flexible delivery. CSM provides solutions for:

1) **Authentication** - validating and verifying a user's credentials to allow access to an application. CSM, working with credential providers (Lightweight Directory Access Protocol (LDAP), Relational Database Management Systems (RDBMS), etc.), confirms that a user exists and that the password is valid for that application.
2) **Authorization** - granting access to data, methods, and objects. CSM incorporates an Authorization schema and database so that users can only perform the operations or access the data to which they have access rights.
3) **User Provisioning** - creating or modifying users and their associated access rights to your application and its data. CSM provides a web-based UPT that can easily be integrated with a single or multiple applications and authorization databases. The UPT provides functionality to create authorization data elements like Roles, Privileges,

Protection Elements, Users, etc., and also provides functionality to associate them with each other. The runtime API can then use this authorization data to authorize user actions. The UPT consists of two modes – Super Admin and Admin.

a. **Super Admin** – accessed by the UPT's overall administrator; used to register an application and assign administrators.
b. **Admin** – used by application administrators to modify authorization data, such as roles, privileges, users, etc.

Figure 2-1 shows how CSM works with an application and independent entities, such as the credential providers and authorization schema, to perform authentication and authorization.

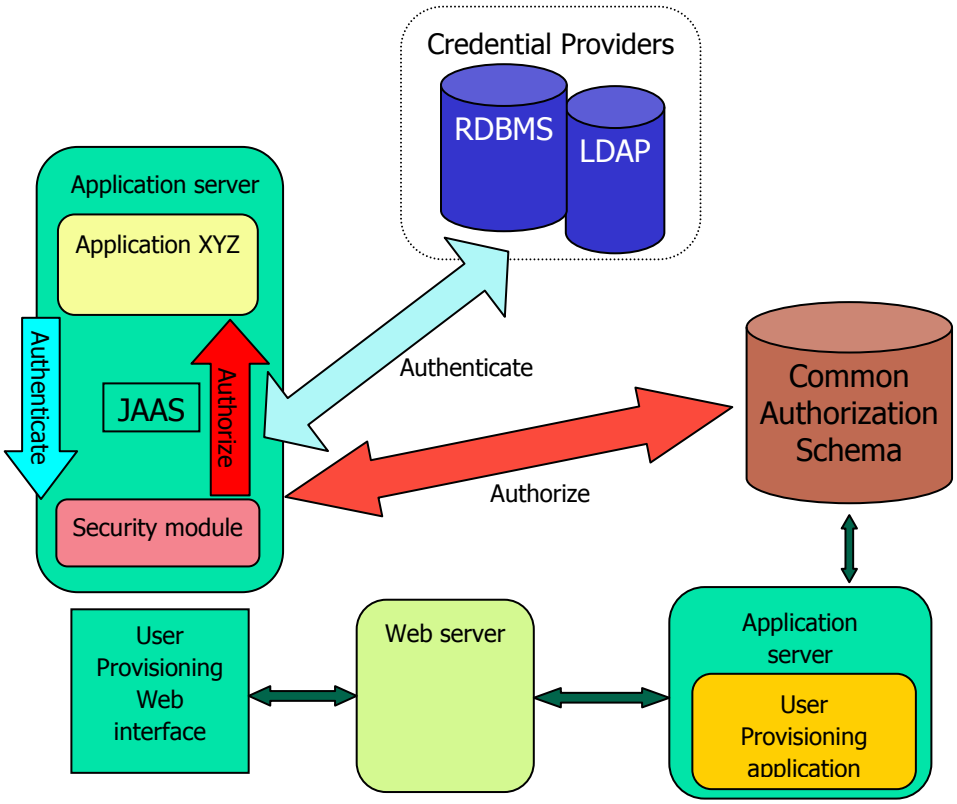

Figure 2-1 CSM interactions for authentication and authorization (see text)

CSM works with Java Authentication and Authorization Service (JAAS) to authenticate and authorize for the Application ABC. To authenticate, it references credential providers such as an LDAP or RDBMS. CSM can be configured to check multiple credential providers in a defined order. To authorize, CSM refers to the Authorization Schema. The Authorization Schema contains the Users, Roles, Protection Elements, etc., and their associations, so that the application knows whether or not to allow a user to access a particular object. The Authorization data can be stored on a variety of databases. It is created and modified by the Application Administrator using the web-based UPT.

2.2 Security Concepts

In order to successfully integrate CSM with an application, it is important to understand the definitions for the security concepts defined in Table 2-1. Application Developers should understand these concepts and begin to understand how they apply to their particular application.

Security Concept	*Definition*
Application	Any software or set of software intended to achieve business or technical goals.
User	A User is someone that requires access to an application. Users can become part of a Group, and can have an associated Protection Group and Roles.
Group	A Group is a collection of application users. By combining users into a Group, it becomes easier to manage their collective roles and access rights in your application.
Protection Element	A Protection Element is any entity (typically data) that has controlled access. Examples include Social Security Number, City, and Salary. Protection Elements can also include operations, buttons, links, etc.
Protection Group	A Protection Group is a collection of application Protection Elements. By combining Protection Elements into a Protection Group, it becomes easier to associate Users and Groups with rights to a particular data set. Examples include Address and Personal Information.
Privilege	A Privilege refers to any operation performed on data. Examples include Delete Record or Modify Record.
Role	A Role is a collection of application Privileges. Examples include Record Admin. and HR Manager.

Table 2-1 Security concept definitions

CSM users need to identify aspects of the application that should be labeled as Protection Elements. These elements are combined to Protection Groups, and then users are assigned Roles for that Protection Group.

Shown in Table 2-2 are definitions of related security terms.

Related Concept	*Definition*
Credential Provider	A credential is a data or set of data which represents an individual uniquely to a given application (username, password, etc.). Credential providers are trusted organizations that create secure directories or databases that store credentials. In an

Related Concept	Definition
	authentication transaction, organizations check with the credential providers to verify entered information is valid. For example, the NCI network uses a credential provider to verify that a user name and password match and are valid before allowing access.
LDAP	Credential providers may choose to store credential information using a directory based on LDAP. An LDAP is simply a set of protocols for accessing information directories. Using LDAP, client programs can login to a server, access a directory, and verify credential entries.
RDBMS	Credential providers may choose to store credential information with a RDBMS. Unlike with LDAP, credential data is stored in the form of related tables.

Table 2-2 Related security concept definitions

3. The Three Services

The Security APIs consist of three primary components - Authentication, Authorization and User Provisioning. The following three corresponding managers control these components:

- AuthenticationManager
- AuthorizationManager
- UserProvisioningManager

3.1 AuthenticationManager

The AuthenticationManager is an interface that authenticates a user against a credential provider. See *Integrating with the CSM Authorization Service* on page 31 to learn how to integrate with the AuthenticationManager.

The AuthenticationManager contains the methods as shown in Table 3-1.

Return type	Method Name	
boolean	**login**(String userName, String password) throws CSException	
Void	**initialize**(String applicationContextName);	
Void	**setApplicationContextName**(String applicationContextName);	
String	**getApplicationContextName**()	
Object	**getAuthenticatedObject**()	(future release)
Subject	**getSubject**()	(future release)

Table 3-1 AuthenticationManager methods

Developers will work primarily with the login method. Detailed descriptions about each method's functionality and its parameters are present in the Javadocs.

3.2 AuthorizationManager

The AuthorizationManager is an interface which provides run-time methods with the purpose of checking access permissions and provisioning certain authorization data. See *Integrating with the CSM Authorization Service* on page 31 to learn how to integrate with the AuthorizationManager.

The AuthorizationManager contains the methods as shown in Table 3-2.

Return Type	*Method*
User	**getUser**(String loginName)
ApplicationContext	**getApplicationContext**()
void	**assignProtectionElement**(String protectionGroupName, String protectionElementObjectId, String protectionElementAttributeName)throws CSTransactionException;
void	**setOwnerForProtectionElement**(String protectionElementObjectId, String[] userNames)throws CSTransactionException;
void	**deAssignProtectionElements**(String protectionGroupName,String protectionElementObjectId)throws CSTransactionException;
void	**createProtectionElement**(ProtectionElement protectionElement)throws CSTransactionException;
boolean	**checkPermission**(AccessPermission permission, Subject subject) throws CSException;
boolean	**checkPermission**(AccessPermission permission, String userName) throws CSException;
boolean	**checkPermission**(String userName, String objectId, String attributeName, String privilegeName) throws CSException;
boolean	**checkPermission**(String userName, String objectId, String privilegeName) throws CSException;
Principal[]	**getPrincipals**(String userName);
ProtectionElement	getProtectionElement(String objectId)throws CSObjectNotFoundException;
ProtectionElement	**getProtectionElementById**(String protectionElementId) throws CSObjectNotFoundException;
void	**assignProtectionElement**(String protectionGroupName, String protectionElementObjectId)throws CSTransactionException;
void	**setOwnerForProtectionElement**(String userName, String protectionElementObjectId, String protectionElementAttributeName)throws CSTransactionException;
void	**initialize**(String applicationContextName);
List	**getProtectionGroups**();
ProtectionElement	**getProtectionElement**(String objectId,String attributeName);

Return Type	Method
Object	**secureObject**(String userName, Object obj) throws CSException;
Collection	**secureCollection**(String userName,Collection objects) throws CSException;
Set	**getProtectionGroups**(String protectionElementId) throws CSObjectNotFoundException;
Collection	**getPrivilegeMap**(String userName,Collection protectionElements) throws CSException;

Table 3-2 AuthorizationManager methods

Detailed descriptions about each method's functionality and its parameters are present in the Javadocs.

3.3 UserProvisioningManager

The UserProvisioningManager is the interface used by the UPT. This manager provides an interface where application developers can provision user access rights. Since the UserProvisioningManager is only used internally by the UPT Tool, it is not discussed in detail in this section.

4. Deployment Models

4.1 Authentication

4.1.1 Introduction

The CSM Authentication Service provides a simple and comprehensive solution for user authentication. Developers can easily incorporate the service into their applications with simple configuration and coding changes. This service allows authentication using LDAP and RMDBS credential providers.

4.1.2 Purpose

This section serves as a guide to help caCORE developers integrate existing applications with the CSM application. This section outlines a step by step process that addresses what developers need to know in order to successfully integrate, including:

- Jar placement
- Configuring the ApplicationSecurityConfig.xml
- Database properties and configuration
- LDAP properties and configuration

4.1.3 Scope

The CSM Authentication Service is available for all caCORE applications. Although it can be used exclusively and is effective on its own, it does not need to replace existing authentication. Rather, it can be used to supplement your application's current authentication mechanism. Currently, only RDBMS-based and LDAP-based authentication is supported by CSM.

4.1.4 Definitions, Acronyms, and Abbreviations

Shown in Table 4-1 are definitions important for understanding the rest of the section.

Term	Definition
ABC Application	In a few instances we refer to an ABC application (abcapp) which is simply a sample application. Use of this example helps to illustrate how to integrate an application in CSM. It has been integrated with the CSM code to perform the authentication using the ABC database.
Login Module	Responsible for authenticating users and for populating users and groups. A Login Module is a required component of an authentication provider, and can be a component of an identity assertion provider if you want to develop a separate LoginModule for perimeter authentication. LoginModules that are not used for perimeter authentication also verify the proof material submitted (for example, a user password).
JAAS	Set of Java packages that enable services to authenticate and enforce access controls upon users. JAAS implements a Java version of the standard Pluggable Authentication Module framework, and supports user- based authorization.

Table 4- 1Definitions for important terms

4.1.5 JAR Placement

The CSM Application is available as a JAR which needs to be placed in the classpath of the application. Along with this JAR, there are many supporting JARs on which the CSM API depends. These should be added in the folder `<application-web-root>\WEB-INF\lib.`

4.1.6 Authentication Properties and Configuration

4.1.6.1 Requirements

If preferred, the client application `abcapp` can use its own AuthenticationManager instance instead of the default JAAS implementation. In order to configure its own implementation of the AuthenticationManager, the client application needs its own entry in the `ApplicationSecurityConfig.xml` file. If no entry is found for the given application context name in the `Authentication.Properties` file, then the default JAAS implementation is used for performing the authentication.

4.1.6.2 Configuring an Authentication Manager

Developers can specify their own AuthenticationManager implementation class by making an entry in `ApplicationSecurityConfig.xml` against the application context name as shown in Figure 4-1. Note that the application name must match the application context name provided at the time of obtaining the instance of the AuthenticationManager using the SecurityServiceProvider. Also the class name provided should be fully qualified.

```
<application>
        <context-name>
                FooApplication
        </context-name>
        <authentication>
                <authentication-provider-class>
                        com.Foo.AuthenticationManagerClass
                </authentication-provider-class>
        </authentication>
                :
                :
</application>
```

Figure 4-1 Specifying AuthenticationManager implementation class

1. The location of the ApplicationSecurityConfig.xml needs to be specified to the API. This is done using a system property. In JBoss, edit the JBoss properties-service.xml to provide a startup parameter to the JBoss server. This file is located at the following path: {jboss-home}/server/standard/deploy/properties-service.xml where {jboss-home} is the base directory where JBoss is installed on the server.
2. Add the following entry to the existing properties in the properties-service.xml file:

   ```
   <attribute name="Properties"> <!-- could already exist -->
   :
   gov.nih.nci.security.configFile=/foo/bar/ApplicationSecurityConfig.xml
   :
   </attribute> <!-- could already exist -->
   ```

   The `gov.nih.nci.security.configFile` is the name of the property which points to the fully qualified path `foo/bar/ApplicationSecurityConfig.xml` where the `ApplicationSecurityConfig.xml` has been created above. The name of the property has to be the `gov.nih.nci.security.configFile` and cannot be modified as it is a system-wide property.
3. Save this file in a deploy folder (for example, `{jboss-home}/server/default/deploy/`)

Note: When deploying to JBoss 3.2.3, the `properties-service.xml` file is already located in the folder: `{jboss-home}/server/default/deploy/`.

4.1.7 Database Properties and Login Module Configuration

4.1.7.1 Requirements

In order to authenticate using the RDBMS database, developers must provide:

- The details about the database

- The actual query which will make the database calls

The CSM goal is to make authentication work with any compatible application or credential provider. Therefore we use the same Login Modules to perform authentication, and these must possess a standard set of properties.

The properties needed to establish a connection to the database include:

- **Driver** - The database driver loaded in memory to perform database operations
- **URL** - The URL used to locate and connect to the database
- **User** - The user name used to connect to the database
- **Password** - The password used to connect to the database

The following property provides the query to be used for the database to retrieve the user.

- **Query** - The query which will be fired against the RDBMS tables to verify the user id and the password passed for authentication

The *Configuring a Login Module in JAAS* section on this page shows how to configure using JAAS or the JBoss `login-config.xml` file.

4.1.7.2 Configuring a Login Module in JAAS

Developers can configure a login module for each application by making an entry in the JAAS configuration file for that application name or context.

The general format for making an entry into the configuration files is shown in Figure 4-2.

```
Application 1 {
          ModuleClass  Flag    ModuleOptions;
          ModuleClass  Flag    ModuleOptions;
          ...
      };
Application 2 {
          ModuleClass  Flag    ModuleOptions;
          ...
      };
      ...
```

Figure 4-2 Configuring a login module

For `abcapp`, which uses RDBMSLoginModule, the JAAS configuration file entry is shown in Figure 4-3.

```
abcapp
{
     gov.nih.nci.security.authentication.loginmodules.RDBMSLoginModule Required
     driver="oracle.jdbc.driver.OracleDriver" url="jdbc:oracle:thin:@cbiodb2-
     d.nci.nih.gov:1521:cbdev"
     user="USERNAME"
     passwd="PASSWORD"
     query="SELECT * FROM users WHERE username=? and password=?"
}
```

Figure 4-3 abcapp JAAS configuration file entry

The configuration file entry contains the following:

- The application is `abcapp`.
- The ModuleClass is `gov.nih.nci.abcapp.loginmodules.RDBMSLoginModule`.
- The `Required` flag indicates that authentication using this credential source is a must for overall authentication to be successful.
- The ModuleOptions are a set of parameters which are passed to the ModuleClass to perform its actions. In the prototype, the database details as well as the query are passed as parameters: `driver="oracle.jdbc.driver.OracleDriver" url="jdbc:oracle:thin:@cbiodb2-d.nci.nih.gov:1521:cbdev" user="USERNAME" passwd="PASSWORD" query="SELECT * FROM users WHERE username=? and password=?"`

Since `abcapp` has only one credential provider only one corresponding entry was made in the configuration file. If the application uses multiple credential providers then the LoginModules can be stacked. A single configuration file can contain entries for multiple applications.

4.1.8 Configuring a Login Module in JBoss

If an application uses the JBoss Server, developers can perform login module configuration differently. Rather than creating a JAAS configuration file, simply use the JBoss `login-config.xml` file which is located at `{jboss-home}\server\{server-name}\conf\login-config.xml`.

Shown in Figure 4-4 is the entry for the `abcapp` application:

```
<application-policy name = "abcapp">
       <authentication>
               <login-module code = "gov.nih.nci.security.loginmodules.RDBMSLoginModule" flag
= "required" >
                       <module-option name="driver"> oracle.jdbc.driver.OracleDriver</module-
option>
                       <module-option name="url">jdbc:oracle:thin:@cbiodb2-
d.nci.nih.gov:1521:cbdev</module-option>
                       <module-option name="user">USERNAME</module-option>
                       <module-option name="passwd">PASSWORD</module-option>
                       <module-option name="query">SELECT * FROM users WHERE username=? and
password=?</module-option>
               </login-module>
       </authentication>
</application-policy>
```

Figure 4-4 Example abcapp entry in login-config.xml

As shown in this example:

- The `application-policy` specifies the application for which we are defining the authentication policy which is `abcapp`.
- The login-module is the LoginModule class which is to be used to perform the authentication task; in this case it is `gov.nih.nci.security.loginmodules.RDBMSLoginModule`.
- The `flag` provided is "`required`".
- The `module-options` list down the parameters which are passed to the LoginModule to perform the authentication task. In this case they are:

  ```
  <module-option
     name="driver">oracle.jdbc.driver.OracleDriver</module-option>

  <module-option name="url">jdbc:oracle:thin:@cbiodb2-
     d.nci.nih.gov:1521:cbdev</module-option>

  <module-option name="user">USERNAME</module-option>

  <module-option name="passwd">PASSWORD</module-option>

  <module-option name="query">SELECT * FROM users WHERE username=?
     and password=?</module-option>
  ```

4.1.9 LDAP Properties and Login Module Configuration

4.1.9.1 Requirements

The default implementation also provides an LDAP-based authentication module to be used by the client applications. In order to authenticate using the LDAP, developers must provide:

- The details about the LDAP server
- The label for the user ID Common Name (CN) or User Identification (UID) in the LDAP server

The properties needed to establish a connection to the database include:

- **ldapHost** – The URL of the actual LDAP server.
- **ldapSearchableBase** – The base of the LDAP tree from where the search should begin.
- **ldapUserIdLabel** – The actual user id label used for the CN entry in LDAP.

4.1.9.2 Configuring a LDAP Login Module in JAAS

For `abcapp`, which uses LDAPLoginModule, the JAAS config file entry is shown in Figure 4-5.

```
abcapp
{
     gov.nih.nci.security.authentication.loginmodules.LDAPLoginModule Required
     ldapHost= “ldaps://ncids2b.nci.nih.gov:636”
     ldapSearchableBase= “ou=nci,o=nih”
     ldapUserIdLabel=”cn”
}
```

Figure 4-5 Example JAAS configuration file entry

As shown in Figure 4-5:

- The application is `abcapp`.
- The ModuleClass is `gov.nih.nci.abcapp.loginmodules.LDAPLoginModule`.
- The `Required` flag indicates that authentication using this credential source is a must for overall authentication to be successful.
- The LDAP details are passed:

  `ldapHost="ldaps://ncids2b.nci.nih.gov:636"`

  `ldapSearchableBase= “ou=nci,o=nih”`

  `ldapUserIdLabel=”cn”`

Note: Since `abcapp` has only one credential provider, only one corresponding entry was made in the configuration file. If the application uses multiple credential providers then the LoginModules can be stacked. A single configuration file can contain entries for multiple applications.

4.1.9.3 Configuring a LDAP Login Module in JBoss

If an application uses the JBoss Server, developers can perform login module configuration differently. Rather than creating a JAAS configuration file, simply use the JBoss `login-config.xml` file which is located at `{jboss-home}\server\{server-name}\conf\login-config.xml`.

Shown in Figure 4-6 is the entry for the `abcapp` application:

```
<application-policy name = "abcapp">
  <authentication>
    <login-module code = "gov.nih.nci.security.loginmodules.LDAPLoginModule" flag =
"required" >
      <module-option name="ldapHost">ldaps://ncids2b.nci.nih.gov:636</module-option>
      <module-option name="ldapSearchableBase">ou=nci,o=nih</module-option>
      <module-option name="ldapUserIdLabel">cn</module-option>
    </login-module>
  </authentication>
</application-policy>
```

Figure 4-6 Example LDAP JBoss configuration file

As shown in Figure 4-6:

- The `application-policy` is the application for which we are defining the authentication policy – in this case `abcapp`.
- The login-module is the LoginModule class which is to be used to perform the authentication task; in this case it is `gov.nih.nci.security.loginmodules.LDAPLoginModule`.
- The `flag` provided is "`required`".
- The `module-options` list down the parameters which are passed to the LoginModule to perform the authentication task. In this case they are:

  `<module-option name="ldapHost">ldaps://ncids2b.nci.nih.gov:636</module-option>`

  `<module-option name="ldapSearchableBase">ou=nci,o=nih</module-option>`

  `<module-option name="ldapUserIdLabel">cn</module-option>`

4.2 Authorization

4.2.1 Introduction

The security APIs have been provided to facilitate the security needs at run time. These APIs can be used programmatically. They have been written using Java, so it is assumed that developers know the Java language.

This section outlines integration steps. For further support, CSM recommends reviewing the CSM Enterprise Architecture Model (found on the NCICB Intranet site: http://ncicbintra.nci.nih.gov/intra/caCORE/documentation).

4.2.2 Software Products

Table 4.2 displays descriptions of software products used for authorization.

Software Product	Description
JBoss	The JBoss/Server is the leading Open Source, standards-compliant, J2EE based application server implemented in 100% Pure Java. A majority of caCORE applications use this server to host their applications.
MySQL	MySQL is an open source database. Its speed, scalability and reliability make it a popular choice for Web developers. CSM recommends storing authorization data in a MySQL database.
Hibernate	Hibernate is an object/relational persistence and query service for Java. CSM requires developers to modify a provided Hibernate configuration file (`hibernate.cfg.xml`) in order to connect to the appropriate application authorization schema.

Table 4- 2 Authorization software products

4.2.3 Integrating CSM APIs – Overview

This section provides instruction for integrating the CSM APIs with JBoss. The integration is flexible enough to meet the needs for several scenarios depending on the number of applications hosted on JBoss and whether or not a common schema is used. Following are the scenarios:

1. JBOSS is hosting a number of applications
   a. use common schema
   b. use separate schema
2. JBOSS is hosting only one application
   a. use common schema
   b. use separate schema

4.2.3.1 Jar Placement

The CSM Application is available as a JAR which needs to be placed in the classpath of the application. Along with this JAR, there are many supporting JARs on which the CSM API depends. These should be added in the folder `<application-web-root>\WEB-INF\lib`.

4.2.4 Deployment Steps

Step 1: Create and Prime a MySQL Database

Note: When deploying Authorization, application developers may want to make use of a previously installed common Authorization Schema. In this case a MySQL database already exists, so skip this step. Also note that the Authorization Schema used by the run-time API and the UPT has to be the same.

1. Log into the MySQL database using an account id which has permission to create new databases.
2. In the AuthorizationSchema.sql file replace the "<<database_name>>" tag with the name of the authorization schema (for e.g. "caArray").
3. Run this AuthorizationSchema.sql on the MySQL prompt. This should create a database with the given name.
4. Now in the AuthorizationSchemaPriming.sql, replace the "<<application_context_name>>" with the name of application. This is the key to derive security for the application. This will be called application context name.
5. Run this AuthorizationSchemaPriming.sql on the MySQL prompt. This should populate the database with the initial data. Verify this by querying the application table. It should include one record only.

Step 2: Configure Datasource

1. Modify the provided `mysql-ds.xml` file which contains information for creating a datasource. One entry is required for each database connection. Edit this file to replace:
   a. the <<application_context_name>> tag with the name of the authorization schema (for example, "csmupt").
   b. the <<database_user_id>> with the user id and <<database_user_password>> with the password of the user account, which will be used to access the Authorization Schema created in Step 1 above.
   c. the <<database_url>> with the URL needed to access the Authorization Schema residing on the MySQL database server.

   Shown in Figure 4-7 is an example of the file.

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>

  <local-tx-datasource>
    <jndi-name>csmupt</jndi-name>
    <connection-url>jdbc:mysql://cbiodev104.nci.nih.gov:3306/csmupt</connection-url>
    <driver-class>org.gjt.mm.mysql.Driver</driver-class>
    <user-name>name</user-name>
    <password>password</password>
  </local-tx-datasource>

  <local-tx-datasource>
    <jndi-name>security</jndi-name>
    <connection-url>jdbc:mysql://cbiodev104.nci.nih.gov:3306/csd</connection-url>
    <driver-class>org.gjt.mm.mysql.Driver</driver-class>
    <user-name>name</user-name>
    <password>password</password>
   </local-tx-datasource>

</datasources>
```

Figure 4-7 Example mysql-ds.xml file

2. Place the `mysql-ds.xml` file in the JBoss deploy directory.

Step 3: Create a Directory

1. Create a directory on the server where all the configuration files pertaining to the application will be kept. This directory can have any name and can reside anywhere on the server. However, it should be accessible to the JBoss id running the application.

Note: In case the application is deployed on a shared server which hosts other applications that are already using CSM, then this folder may already exist.

Step 4: Configure Hibernate

1. The provided hibernate.cfg.xml file requires modification to include configuration details to connect to the appropriate application authorization schema. For the property connection.datasource, replace the <<upt_context_name>> with the application name for the UPT. For example, the property may contain java:/security or java:/caArray. This application name should be same as the one created in Step 1.

2. Rename this file as <<application_context_name>>.hibernate.cfg.xml (e.g. for caArray it will be caArray.hibernate.cfg.xml). Place this file in the directory created in Step 2. Make sure that the JBoss id has access to it.

Note: If the application requires use of a commonly installed Authorization Schema, it can use the same Hibernate configuration.

Step 5: Modify ApplicationSecurityConfig.xml

1. Edit the provided `ApplicationSecurityConfig.xml` as shown in Figure 4-8. Replace the `<<application_context_name>>` with the application name. This application name should be the same as the one created in Step 1.

```
<application>
      <context-name>
         <<application_context_name>>
      </context-name>
         <authentication>
            <authentication-provider-class>
                <!-- Fully qualified class name-->
              </authentication-provider-class>
          </authentication>
          <authorization>
              <authorization-provider-class>
               <!-- Fully qualified class name-->
             </authorization-provider-class>
            <hibernate-config-file>
               <!-- Fully qualified file path -->
                    <<hibernate_cfg_file_path>>
            </hibernate-config-file>
          </authorization>
</application>
```

Figure 4-8 Example ApplicationSecurityConfig.xml file

2. Also edit the file to replace the <<hibernate_cfg_file_path>> with the fully qualified path of the hibernate configuration file <<application_context_name>>.hibernate.cfg.xml (for example, for caArray it will be caArray.hibernate.cfg.xml created in Step 4).

3. Place this file in the directory mentioned in Step 3. Make sure that the JBoss id has access to it.

Note: If the application is deployed on a shared server which hosts other applications that are already using CSM, then this file may be present already. Note that there can only be one `ApplicationSecurityConfig.xml` file per JBoss installation, so simply add a new application entry to the existing file.

Step 6: Make an Addition to the JBoss Startup Properties File

1. Edit the `JBoss properties-service.xml` to provide a startup parameter to the JBoss server. This file is located at `{jboss-home}/server/standard/deploy/properties-service.xml` where `{jboss-home}` is the base directory where JBoss is installed on the server. Add the following entry:

```
<attribute name="Properties"> <!-- could already exist -->
:
gov.nih.nci.security.configFile=/foo/bar/ApplicationSecurityConfig.xml
:
</attribute> <!-- could already exist -->
```

- The `gov.nih.nci.security.configFile` is the name of the property which points to the fully qualified path `foo/bar/ApplicationSecurityConfig.xml` where the `ApplicationSecurityConfig.xml` was created in Step 4. The name of the property has to be the `gov.nih.nci.security.configFile` and cannot be modified as it is a system-wide property.

2. Save this file in a deploy folder. An example is: `{jboss-home}/server/default/deploy/.`

Note: When deploying to JBoss 3.2.3, the `properties-service.xml` file is already located in the folder `{jboss-home}/server/default/deploy/.` In case the application is deployed on a shared server which hosts other applications that are already using CSM, then this property could be present.

4.3 Provisioning

4.3.1 Introduction

UPT is a web application used to provision an application's authorization data. The UPT provides functionality to create authorization data elements like Roles, Privileges, Protection Elements, Users, etc., and also provides functionality to associate them with each other. The runtime API can then use this authorization data to authorize user actions.

This section of the guide explains how to deploy the UPT from start to finish - from uploading the Web Application Archive (WAR) and editing configuration files, to synching the UPT with the application. See *Integrating with the User Provisioning Service* on page 32 if you need to integrate with an existing UPT deployment.

This section details the UPT release contents, explains multiple ways in which the UPT can be deployed, and outlines the steps that result in a successful deployment.

4.3.2 UPT Release Contents

The UPT is released as a compressed web application in the form of a WAR (Web Archive) File. Along with the WAR, the release includes sample configuration files that help developers configure the UPT with their application(s).

The UPT Release contents can be found in the `UPT.zip` file found on the NCICB download site (http://ncicb.nci.nih.gov/download/index.jsp). The UPT Release contents include the files in Table 4.3.

File	Description
csmupt.war	The UPT Web Application
ApplicationSecurityConfig.xml	The XML file containing the configuration data for the UPT.
hibernate.cfg.xml	The sample XML file which contains the hibernate-mapping and the database connection details.

File	Description
AuthorizationSchema.sql	This Structured Query Language (SQL) script is used to create an instance of the Authorization database schema which will be used by UPT for the purpose of authorization. The same script can be used to create instances of authorization schema for a variety of applications.
UPTDataPriming.sql	This SQL script is used for priming data in the UPT's authorization schema.
mysql-ds.xml	This file contains information for creating a datasource. One entry is required for each database connection. Place this file in the JBoss deploy directory.

Table 4- 3 UPT release contents

4.3.3 UPT Installation Modes

UPT was developed as a flexible application that can be deployed in multiple ways depending on the need or scenario. The three primary modes to install the UPT include the following and are described in the following sections:

- Single Installation, Single Schema
- Single Installation, Multiple Schemas
- Local installation, Local schema

4.3.3.1 Single Installation, Single Schema

In the single installation, single schema deployment scheme as shown in Figure 4-9, there is only one instance of UPT hosted on a Common JBoss Server. A common installation is used to administer the authorization data for all applications. The authorization data for all the applications is stored on a common database. Therefore an application using UPT does not have to install its own authorization schema. Also, all applications can use the same `hibernate-config` file since they point to the same database.

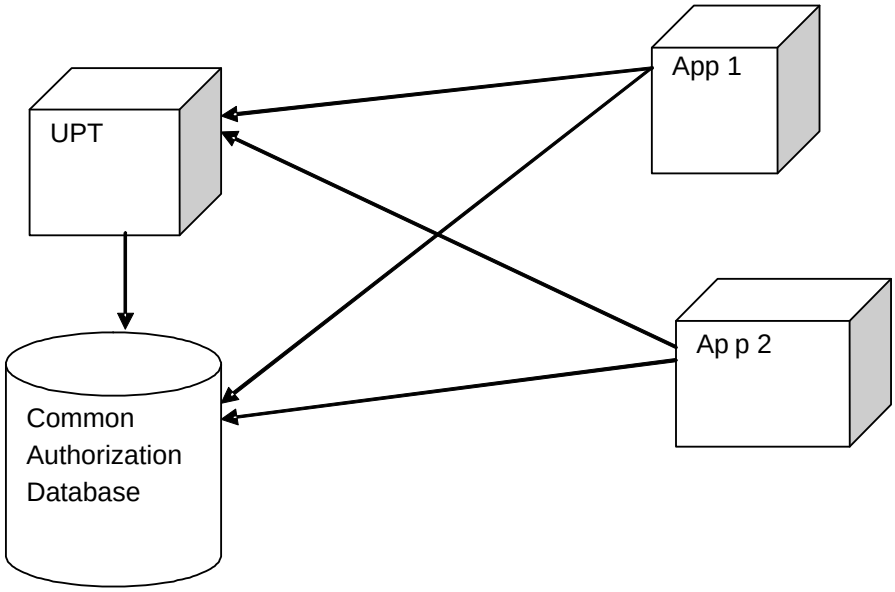

Figure 4-9 Single installation, single schema deployment scheme

4.3.3.2 Single Installation, Multiple Schemas

As in the single schema deployment, the single installation, multiple schemas deployment calls for the UPT to be hosted on a single JBoss Common Server as shown in Figure 4-10. A common installation is also used to administer the authorization data for all applications. What makes this mode different is that an application can use its own authorization schema on a separate database if preferred. The authorization data can sit on individual databases, and at the same time some applications can still opt to use the Common Authorization Schema. Using this mode requires each application to maintain its own `hibernate-config` file pointing to the database where its Authorization Schema is located. So when an application uses the UPT, the UPT communicates to the authorization schema of that application only.

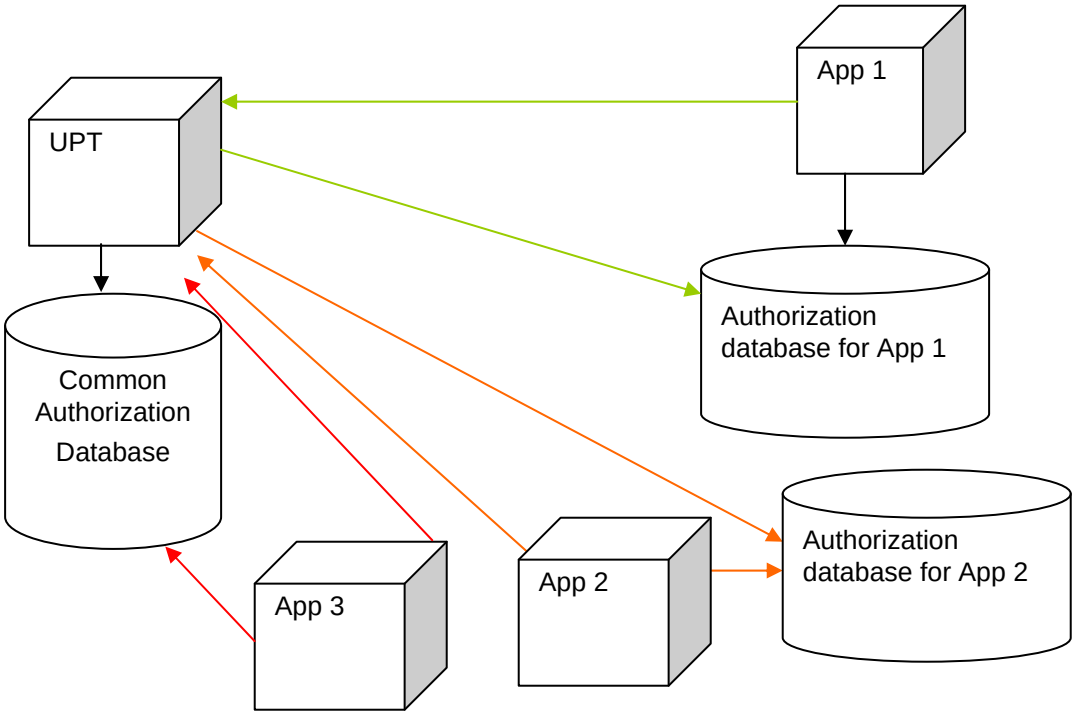

Figure 4-10 Single installation, multiple schemas deployment scheme; the three colors of arrows correspond to the three different applications shown

4.3.3.3 Local Installation, Local Schema

The local installation, local schema deployment is the same as single installation, single schema, except that the UPT is hosted locally by the application as shown in Figure 4-11. This installation of UPT is not shared with other applications. This local installation is used to administer the authorization data for that particular application (or set of related applications) only. The authorization data for the application sits on its own database. In this scenario, the application requires its own `hibernate-config` file pointing to the database where its Authorization Schema is located.

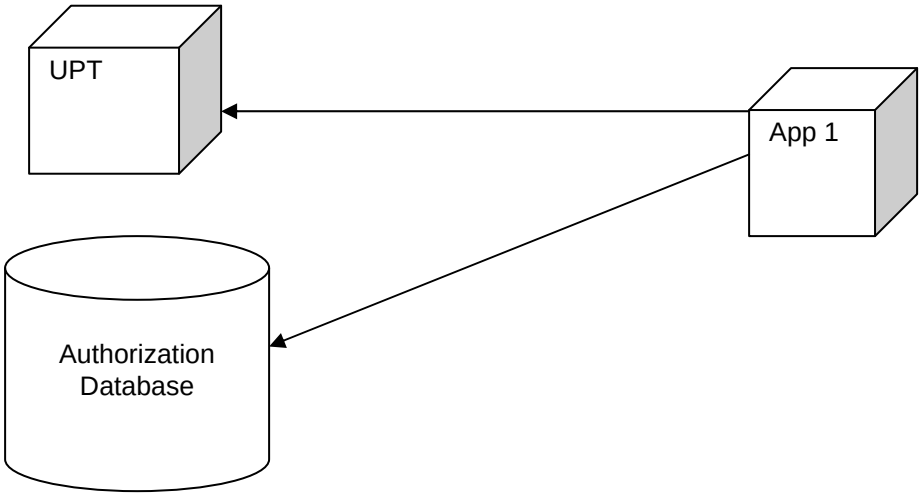

Figure 4-11 Local installation, local schema deployment scheme

4.3.4 Deployment Checklist

Before deploying the UPT, verify the following environment and configuration conditions are met. This software and access credentials/parameters are required.

- Environment
  - JBoss 4.0 Application Server
  - MySQL 4.0 Database Server (with an account that can create databases)
- UPT Release Components
  - csmupt.war
  - AuthorizationSchema.sql
  - UPTDataPriming.sql
  - ApplicationSecurityConfig.xml
  - hibernate-config file

4.3.5 Deployment Steps

Step 1: Create and Prime MySQL Database

1. Log into MySQL database using an account id which has permission to create new databases.
2. In the AuthorizationSchema.sql file replace the <<database_name>> tag with the name of the UPT Authorization schema (for example, “csmupt”).
3. Run this AuthorizationSchema.sql on the MySQL prompt. This should create a database with the given name.
4. In the UPTDataPriming.sql, replace:
   - The <<upt_context_name>> with the name of UPT application. (For simplification, it

is better to name the database and UPT application the same; for example, “csmupt”).

- The <<super_admin_login_id>> with the login id of the user who is going to act as the Super Admin for that particular installation of UPT. If using the NCICB LDAP as the authentication mechanism, the super_admin_login_id should be the NCICB LDAP login id for that particular user. Also provide the first name and last name for the same by replacing <<super_admin_first_name >> and <<super_admin_last_name >>.

5. Run the UPTDataPriming.sql on the MySQL prompt. This should populate the database with the initial data. Verify by querying the application, user, protection_element and user_protection_element tables. They should have one record each.

Step 2: Configure Datasource

1. Modify the mysql-ds.xml file which contains information for creating a datasource. One entry is required for each database connection. Edit this file to replace:

   - The <<application_context_name>> tag with the name of the authorization schema (for example, “csmupt”).
   - The <<database_user_id>> with the user id and <<database_user_password>> with the password of the user account, which will be used to access the Authorization Schema created in Step 1 above.
   - The <<database_url>> with the URL needed to access the Authorization Schema residing on the MySQL database server.

   Shown in Figure 4-12 is an example `mysql-ds.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>

  <local-tx-datasource>
    <jndi-name>csmupt</jndi-name>
    <connection-url>jdbc:mysql://cbiodev104.nci.nih.gov:3306/csmupt</connection-url>
    <driver-class>org.gjt.mm.mysql.Driver</driver-class>
    <user-name>name</user-name>
    <password>password</password>
  </local-tx-datasource>

  <local-tx-datasource>
    <jndi-name>security</jndi-name>
    <connection-url>jdbc:mysql://cbiodev104.nci.nih.gov:3306/csd</connection-url>
    <driver-class>org.gjt.mm.mysql.Driver</driver-class>
    <user-name>name</user-name>
    <password>password</password>
   </local-tx-datasource>

</datasources>
```

Figure 4-12 Example mysql-ds.xml file

2. Place the mysql-ds.xml file in the JBoss deploy directory.

Step 3: Create Directory

1. Create a directory on the server where all the configuration files pertaining to the UPT

will be kept. This directory can have any name and can reside anywhere on the server. However, it should be accessible to the JBoss id running the UPT.

Step 4: Configure Hibernate

1. The provided hibernate.cfg.xml file needs to be modified to include configuration details to connect to the appropriate UPT Authorization Schema. For the property connection.datasource, replace the <<upt_context_name>> with the application name for the UPT. For example, the property may contain java:/upt or java:/csmupt. This application name should be the same as the one created in Step 1.

2. Rename this file to upt.hibernate.cfg.xml (add upt Prefix). Place this file in the directory created in Step 3. Make sure that the JBoss id has access to it.

Step 5: Modify ApplicationSecurityConfig.xml

1. Edit the provided ApplicationSecurityConfig.xml.

2. Replace the <<upt_context_name>> with the application name for the UPT. This application name should be same as the one created in Step 1.

3. Replace the <<hibernate_cfg_file_path>> with the fully qualified path of the hibernate configuration file upt.hibernate.cfg.xml created in Step 3.

4. Place this file in the directory. Make sure that the JBoss id has access to it.

Step 6: Make an Addition to the JBoss Startup Properties File

1. Edit the JBoss properties-service.xml to provide a startup parameter to the JBoss server. This file is located at the following path: {jboss-home}/server/standard/deploy/properties-service.xml where {jboss-home} is the base directory where JBoss is installed on the server. Add the following entry to the existing properties:

```
<attribute name="Properties"> <!-- could already exist -->
:
gov.nih.nci.security.configFile=/foo/bar/ApplicationSecurityConfig.xml
:
</attribute> <!-- could already exist -->
```

   - The `gov.nih.nci.security.configFile` is the name of the property which points to the fully qualified path `foo/bar/ApplicationSecurityConfig.xml` where the `ApplicationSecurityConfig.xml` has been created in Step 4. The name of the property must be the `gov.nih.nci.security.configFile` and cannot be modified, as it is a system-wide property.

2. Save this file in a deploy folder (for example, {jboss-home}/server/default/deploy/).

Note: When deploying to JBoss 3.2.3, the `properties-service.xml` file is already located in the folder `{jboss-home}/server/default/deploy/.`

Step 7: Configure the JBoss JAAS Login Parameters

The suggested Authentication Credential Provider for UPT is NCICB LDAP. In order to configure the UPT to verify against the LDAP, create an entry in the `login-config.xml` of JBoss as shown in Figure 4-13. This entry configures a login-module against the UPT application context. The location of this file is `{jboss-home}/server/default/conf/login-config.xml` where `{jboss-home}` is the base directory where JBoss is installed on the server.

```
<application-policy name = "abcapp">
  <authentication>
    <login-module code =
"gov.nih.nci.security.authentication.loginmodules.LDAPLoginModule"
flag = "required" >
      <module-option
   name="ldapHost">ldaps://ncids2b.nci.nih.gov:636</module-option>
      <module-option name="ldapSearchableBase">ou=nci,o=nih</module-
option>
      <module-option name="ldapUserIdLabel">cn</module-option>
    </login-module>
  </authentication>
</application-policy>
```

Figure 4-13 Example login-config.xml entry

As shown in Figure 4-13:

- The `application-policy` is the name of the application for defining the authentication policy – in this case, "`abcapp`".
- The `login-module` is the `LoginModule` class which is used to perform the authentication task; in this case, it is `gov.nih.nci.security.authentication.loginmodules.LDAPLoginModule`.
- The `flag` provided is "`required`".
- The `module-options` list the parameters which are passed to the `LoginModule` to perform the authentication task. In this case, they are pointing to the NCICB LDAP Server:

```
<module-option name="ldapHost">ldaps://ncids2b.nci.nih.gov:636</module-
   option>
<module-option name="ldapSearchableBase">ou=nci,o=nih</module-option>
<module-option name="ldapUserIdLabel">cn</module-option>
```

Step 8: Deploy the UPT WAR

1. Copy the UPT upt.war in the deployment directory of JBoss which can be found at {jboss-home}/server/default/deploy/ where {jboss-home} is the base directory where JBoss is installed on the server.

Step 9: Start JBoss

1. Once the deployment is completed, start JBoss. Check the logs to confirm there are no

errors while the UPT application is deployed on the server.

2. Once the JBoss server has completed deployment, open a browser to access the UPT. The URL will be http://<<jboss-server>>/upt, where the <<jboss-server>> is the IP or the DNS name of JBoss Server.

3. The UPT Login Page displays. Enter the UPT Application using the login-id that was assigned to the Super Admin in Step 1 and its password. Also use the UPT Application Name specified in Step 4 for the Application Name.

4. You should be able to login successfully and the UPT Application Home Page displays.

Note: In case of any errors, follow a debugging and trouble shooting procedure to diagnose and solve the issues.

Step 10: Add a New Application

Once the initial setup of UPT is complete, UPT is up and available for the applications to start using provisioning for their authorization data. However, applications must be registered and configured before they can start using the UPT.

1. To register an application, use the UPT front end user interface to create an entry for the new application. Login as a Super Admin, go to the Application section, and select Create a New Application. Once the details are entered, go to the User section to create Users. Then return to the Application section to assign these Users as Application Administrators.

2. Once the application registration is complete, it needs to be configured. First, make a new "application" entry in the ApplicationSecurityConfig.xml file. (Use the existing UPT application entry as a template - copy, paste, and modify for the new application.)

   a. Replace the <context-name> with the new application name.

   b. If the Application will use the default CSM provided Authorization Manager, then leave the <authorization-provider-class> blank.

   c. Replace the hibernate-config qualified path to point to the application's hibernate-config file. (Make sure the hibernate-config file resides in the correct location.) If the application is going to use the Common Authorization Schema (which also hosts the Schema for the UPT itself), then it can use the same hibernate-config file. In that case, just copy the entry from UPT's configuration.

5. Integrating with the CSS Authentication Service

5.1 Importing and Using the CSM Authentication Manager Class

To use the CSM Service, add the highlighted import statements (last two) as shown in Figure 5-1 to the action classes that require authentication.

```
import gov.nih.nci.abcapp.UserCredentials;
import gov.nih.nci.abcapp.model.Form;
import gov.nih.nci.abcapp.util.Constants;
import gov.nih.nci.security.SecurityServiceProvider;
import gov.nih.nci.security.AuthenticationManager;
```

Figure 5-1 Example ABC application - Import statements in an action class

The class `SecurityServiceProvider` is the common interface class exposed by the CSM application. It contains methods to obtain the correct instance of the `AuthenticationManager` configured for that application. The client application `abcapp` then uses the `AuthenticationManager` to perform the actual authentication using the CSM.

Figure 5-2 illustrates an example of how to use the `CSMService` class in the ABC application.

```
UserCredentials credentials = new UserCredentials();
credentials.setPassword(Form.getPassword());
credentials.setUsername(Form.getUsername());

//Get the user credentials from the database and login
try {

        AuthenticationManager authenticationManager =
    SecurityServiceProvider.getAuthenticationManager("abcapp");
        boolean loginOK = authenticationManager.login(credentials.getUsername(),
    credentials.getPassword());
        if (loginOK)
        {
          System.out.println(">>>>>>>>>>>>>> SUCESSFUL LOGIN <<<<<<<<< ");
        }
        else
        {
          System.out.println(">>>>>>>>>>>>>> ERROR IN LOGIN <<<<<<<<< ");
        }
    }
catch (CSException cse){
    System.out.println(">>>>>>>>>>>>>> ERROR IN LOGIN <<<<<<<<< ");
    }
```

Figure 5-2 Example code to use the CSMService class in the ABC application

The client class obtains the default implementation of the `AuthenticationManager` by calling the static `getAuthenticationManager` method of the `SecurityServiceProvider` class by passing the application Context name – in this example "`abcapp`". It then invokes the login method - passing the user's ID and password. Please note that application name should match the name used in the configuration files for JAAS to work correctly. If the credentials provided are correct then a Boolean true is returned indicating that the user is authenticated. If there is an authentication error, a `CSException` is thrown with the appropriate error message embedded.

6. Integrating with the CSM Authorization Service

6.1 Importing and Using the CSM Authorization Manager Class

To use the CSM Service, add the highlighted import statements (last two) as shown in Figure 6-1 to the action classes that require authorization.

```
import gov.nih.nci.abcapp.UserCredentials;
import gov.nih.nci.abcapp.model.Form;
import gov.nih.nci.abcapp.util.Constants;
import gov.nih.nci.security.SecurityServiceProvider;
import gov.nih.nci.security.AuthorizationManager;
```

Figure 6-1 Example ABC application - Import statements in an action class

The class `SecurityServiceProvider` is the common interface class exposed by the CSM application. It contains methods to obtain the correct instance of the `AuthorizationManager` configured for that application. The client application `abcapp` then uses the `AuthorizationManager` to perform the actual authentication using the CSM.

Figure 6-2 illustrates an example of how to use the CSMService class in the ABC Application.

```
try {

              AuthorizationManager authorizationManager =
      SecurityServiceProvider.getAuthorizationManager("abcapp");
              boolean hasPermission = authorizationManager.checkPermission("user name" ,
      "resource name", "operation" );
              if (hasPermission)
              {
                System.out.println(">>>>>>>>>>>>> PERMISSION GRANTED <<<<<<<<< ");
              }
              else
              {
                System.out.println(">>>>>>>>>>>>>PERMISSION DENIED <<<<<<<<< ");
              }
      }
catch (CSException cse){
      System.out.println(">>>>>>>>>>>>> ERROR IN AUTHORIZATION <<<<<<<<< ");
      }
```

Figure 6-2 Example code to use the CSMService class in the ABC application

The client class obtains the default implementation of the `AuthorizationManager` by calling the static `getAuthorizationManager` method of the `SecurityServiceProvider` class by passing the application Context name – in this example "`abcapp`". It then invokes the `checkPermission` method – passing the user's ID, the resources which he is trying to access and the operation which it wants to perform. Note that the application name should match the name used in the configuration files as well as configured in the databases for authorization to work correctly. If the user has the required access permission, then a Boolean true is returned indicating that the user is authenticated. In case of any authorization error, a `CSException` is thrown with the appropriate error message embedded.

7. Integrating with the User Provisioning Service

This section's intended audience is developers wishing to integrate their application(s) with an existing UPT Deployment. (For a complete guide to UPT deployment, see *Provisioning* on page 21.)

Once the initial setup of UPT is complete, UPT is up and available for the applications to start using provisioning for their authorization data. However, applications must be registered and configured before you can start using the UPT.

1. To register an application, use the UPT front end user interface to create an entry for the new application. Simply login as a Super Admin, go to the Application section, and select Create a New Application. Once the details are entered, go to the User section to create Users. Then return to the Application section to assign these Users as Application Administrators. (If you're not the Super Admin, ask him/her to add your Application and you as an Admin.)

2. Once the application registration is complete, it needs to be configured. First, make a new "application" entry in the `ApplicationSecurityConfig.xml` file. (Use the existing UPT application entry as a template - copy, paste, and modify for the new application.)
3. Replace the `<context-name>` with the new application name
4. If the Application will use the default CSM provided Authorization Manager then leave the `<authorization-provider-class>` blank.
5. Replace the `hibernate-config` qualified path to point to the application's `hibernate-config` file. (Make sure the `hibernate-config` file resides in the correct location.) If the application is going to use the Common Authorization Schema (which also hosts the Schema for the UPT itself), then it can use the same `hibernate-config` file. In that case, just copy the entry from the UPT's configuration.