IBM Cúram Social Program Management
Version 7.0.0

Cúram Security Guide

Note

Before using this information and the product it supports, read the information in "Notices" on page 51

Edition

This edition applies to IBM Cúram Social Program Management v7.0.0 and to all subsequent releases unless otherwise indicated in new editions.

Contents

Figures

Tables

Configuring security

Authentication and authorization are two key components of application security. The IBM Cúram webclient is configured to support form-based authentication. Different authentication modes can be configured with the Cúram JAAS login module. Functional elements in Cúram are secured by security identifiers. This data is linked to a user and can be configured.

Authentication Overview

In Cúram, authentication is the process of determining if a user is who they say they are. Authentication is needed where a user must be verified in order to access a secure resource on a system.

Form-based authentication is where a user is presented with a form allowing them to enter username and password credentials. These credentials are compared against the credentials stored on the system for this username, if they match the user is considered an authenticated user for the system. For security reasons the password for authenticating a user is stored on the system in a digested form.

The Cúramweb client is configured to support form-based authentication, which means that before a user can access any of the web client content, they will be redirected to a login form to authenticate.

The authentication process involves the verification of the username and password, and this is performed by default by a JAAS (Java™ Authentication and Authorization Service) login module. HTTPS/SSL is turned on by default in the web client ensuring the form-based login authentication mode is secure.

Authentication

Different authentication modes can be configured (depending on authentication requirements) by the Cúram Java Authentication and Authorization Service (JAAS) login module.

The following are the authentication modes supported:
- Default Authentication
- Identity Only Authentication
- External Access Security Authentication

Each of these modes is described in detail in the sections that follow.

Authentication Architecture

Use the information in this flow chart to understand the architecture for the authentication process of a user.

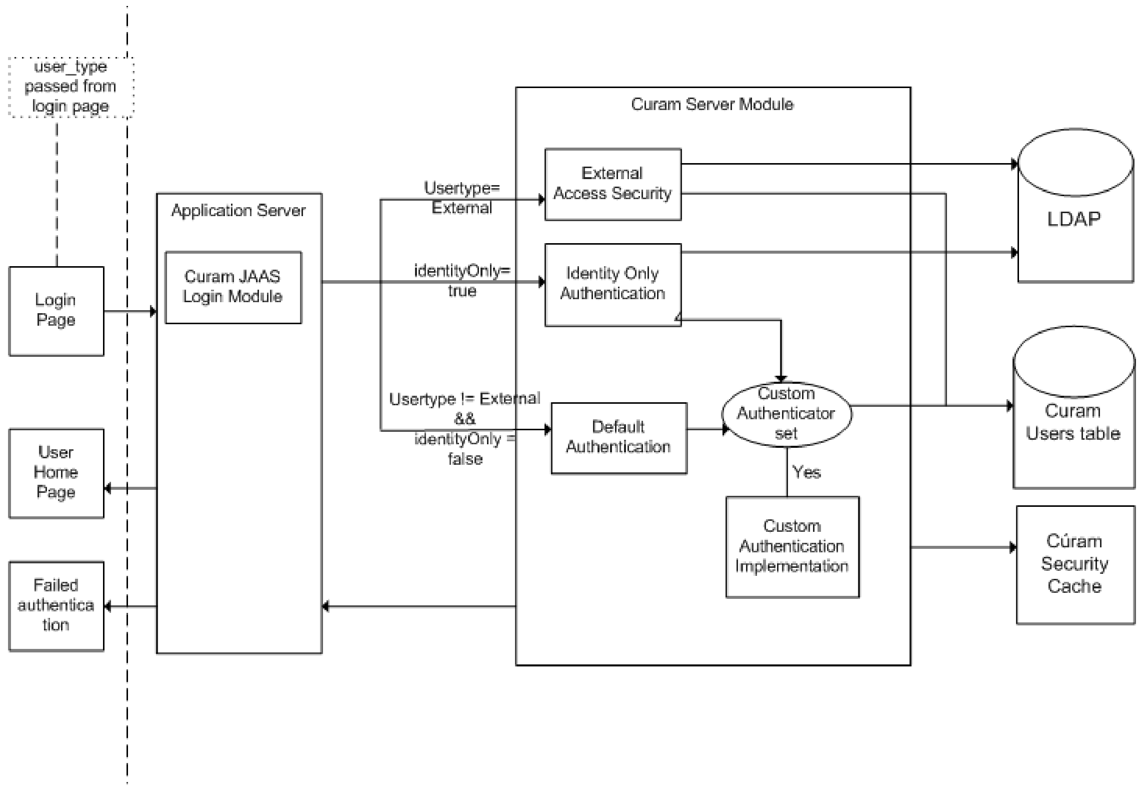

Figure 1. Authentication architecture

The flow chart shown here outlines the architecture for the authentication process of a user. The default authentication is completed for a user. This behavior can be customized for both internal and external users, depending on the authentication requirements. The sections in Authentication Overview chapter that follow describe in detail each of the functional areas that make up the authentication architecture, indicating where customizations are possible.

Default Authentication

Default authentication for Cúram involves the user who logs in through the login screen, where the user is prompted for a `username` and `password` as credentials. These credentials then are passed to the Cúram Java Authentication and Authorization Service (JAAS) login module configured in the application server.

The default authentication is run and the `username` and `password` entered are checked against the `username` and `password` stored on the Cúram Users database table. The Cúram `username` is immutable, but you have the option of configuring your system to use a Cúram `login ID` instead, which is changeable. The `login ID` is a logical extension of the Cúram user and the same verifications that are checked for the `username` also are checked for the `login ID`. For more information about alternate login IDs, see “Alternate Login IDs” on page 3.

Authentication runs a number of verifications against the login credentials. For more information on the login verifications, see “Default Authentication” on page 11.

Provided all verifications are successful, the user is considered to be authenticated by the application.

After the user is authenticated, the user then is added to the Cúram Security Cache. The Cúram Security Cache stores the `username` and all related authorization data for that user to optimize the authorization data retrieval for a user. For more information on the Cúram Security Cache, see "Security Data Caching" on page 21. Figure 2.3 highlights the path taken for default authentication.

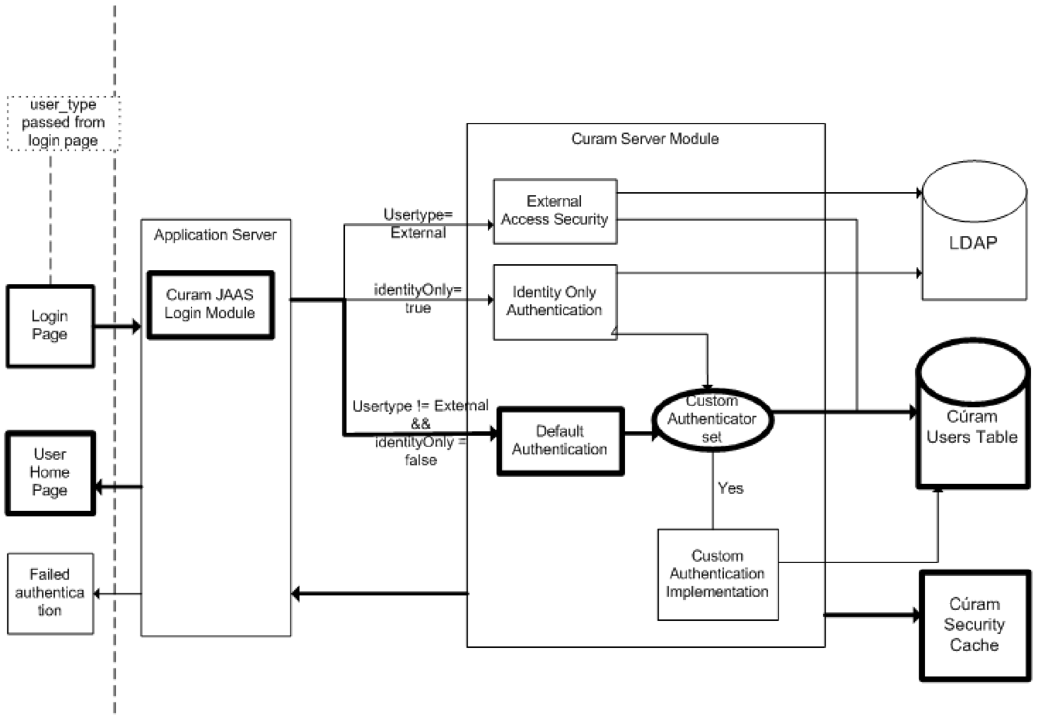

Figure 2. Default authentication

Alternate Login IDs

By default, Cúram uses the `username` and digested `password` that is stored in the `Cúram Users` table for authentication. The `username` cannot be changed after it is created and the lack of flexibility might not meet requirements for some installations. Therefore, you have the option of configuring an alternate `login ID` that can be updated. However, if the default security implementation that is configured during installation meets your requirements, it is not necessary to configure an alternate `login ID`.

The `login ID` functions as a logical extension of the `Cúram Users` table. When the alternate `login ID` is used the `username` still exists and is used internally by Cúram, but the user logs in to Cúram by using the `login ID`.

Things to note when using the alternate `login ID`:

- The Cúram users can log in with their alternate login IDs if available or user names if not. When the property alternate login is disabled, users are only allowed to log in with their user names.
- The Cúram `ExtendedUsersInfo` table, where the `login ID` is stored, must be populated before the application turns on the alternate `login ID` feature, which is explained in more detail in the following explanation.
- When using login IDs, authentication results are stored in the `AuthenticationLog` table and the `AltLogin` column indicates whether the `UserName` column represents a `username` (false) or `login ID` (true).

- Login IDs are only applicable to internal Cúram users; that is, users stored on the `Cúram Users` table. However, if you are using identity-only with alternate Login IDs then wherever those IDs are stored (for example, WebSphere® registry, Lightweight Directory Access Protocol (LDAP), and so on) must match the login IDs stored in the Cúram `ExtendedUsersInfo` table.
- When assigning login IDs, you need to take care with IDs that are used internally or have dependencies (for example, with property values) outside of the `Cúram Users` table. These IDs are the user names that would cause issues if their`login ID` differed from the `username` without a corresponding change as indicated:
  - `SYSTEM` - In WebSphere this user name is associated with Java Message Service (JMS) processing and is made part of the WebSphere configuration at application deployment time. For more information on changing this ID, see "Mandatory Cúram Users" on page 22 and the appropriate WebSphere *Cúram Deployment Guide*.
  - `DBTOJMS` - this value is the default `DBtoJMS username` used by batch processing and is referenced by property `curam.security.credentials.dbtojms.username`. For more information, see "Mandatory Cúram Users" on page 22, "JMS Messaging" on page 23, and"Deferred Processing" on page 24 the *Cúram Batch Processing Guide.*
  - `WEBSVCS` - this value is the default web services `username` and is referenced by property `curam.security.credentials.ws.username`. For more information, see "Mandatory Cúram Users" on page 22, "Web Services" on page 23, and the *Cúram Web Services Guide*.
  - Unauthenticated - is the principal WebSphere uses for unauthenticated users and this `login ID` should not be changed.

To enable the use of the alternate `login ID`, after you have populated the `ExtendedUsersInfo` table, set the `curam.security.altlogin.enabled` property to `true` (for more information on Cúram properties, see the *Cúram Server Developer's Guide*). This value is a static property and Cúram must be restarted for it to take effect.

When the `curam.security.altlogin.enabled` property is set to `true`, authentications are not processed directly through the user name column in the `Cúram Users` table. Instead, authentications are all processed through the `ExtendedUsersInfo login ID`, which references the `Cúram Users` table.

Along with the introduction of support for an alternate login ID, the administrative pages for adding, updating, and displaying Curam users have been extended to include the new **Login ID** field. The **Login ID** field is displayed only when the corresponding `curam.security.altlogin.enabled` property is set to true.

To populate the `ExtendedUsersInfo` table (see table that follows for `ExtendedUsersInfo` you have a number of options; for instance:

- With a simple SQL statement, you can populate the table by using the user name in the `Users` table; so, there is no immediate user impact: `INSERT INTO EXTENDEDUSERSINFO (USERNAME, LOGINID, UPPERLOGINID, VERSIONNO) (SELECT USERNAME, USERNAME, UPPER(USERNAME), 1 FROM USERS);` You can then roll out your modifications to the login IDs in a controlled manner.
- You can implement an SQL application that implements your user name and `login ID` mapping (for example, LDAP common names).

Note: You must maintain the user name foreign key relationship between the `Users` and `ExtendedUsersInfo` tables.

Table 1. `ExtendedUsersInfo` Table Structure

Name	Type	Size	Description
`USERNAME`	VARCHAR	256	`Username` is an immutable string. This field has a foreign key relationship with `username` field in Users table.
`LOGINID`	VARCHAR	1280	`Login ID` is associated to the user name and can be updated. The login ID functions as a logical extension of the `Cúram Users` table. Users can log in to Cúram application by using `Login ID`.
`UPPERLOGINID`	VARCHAR	1280	`Login ID` in uppercase. Uppercase login ID is used for supporting case-insensitivity.
`Version No`	VARCHAR	4	Version Number.

Configuring internal and external users

If you have both internal and external users, extra calls might occur to the `getRegisteredUserName()` method in the `ExternalAccessSecurity` class. The security cache calls the `getRegisteredUserName()` method if the login ID is not found in the security cache. Therefore, all internal and external login IDs and user names must be unique, unless the `curam.util.security.UserScope interface` is implemented. Otherwise, an external user that matches a login ID might be found in the security cache and therefore not found as an external user. If a login ID can't be found either in the cache, or through the External Access Security implementation if it is provided, then an `INFRASTRUCTURE.INFO_LOGIN_ID_DOES_NOT_MAP_TO_USERNAME` exception occurs.

Configuring a custom alternate login implementation

A customer can set the `curam.citizenworkspace.alternate.login.implementation` property to point to a custom alternate login implementation, as shown in the following example:

```
curam.citizenworkspace.alternate.login.implementation=curam.citizenworkspace.security.impl.SampleC
```

A customer can use the alternate login implementation to specify custom code that returns the user name when an alternate login ID is submitted. The alternate login implementation must extend the `CitizenWorkspaceAlternateLogin` abstract class and provide an implementation for the `getRegisteredUsername(final String loginId)` method.

The Login Page

The default preconfigured login page is represented by the `logon.jsp` file. This `logon.jsp` represents the login page for the user to complete form-based login authentication. By default, the `logon.jsp` file contains the `username` and `password` fields.

However, the `logon.jsp` file can be customized to pass an additional parameter by adding the user_type field. This field determines the type of user who is logging in, that is, internal or external user. The `username`, `password`, and `user_type` (if present) are all passed to the Cúram Java Authentication and Authorization Service (JAAS) login module as part of the authentication process.

The default preconfigured `logon.jsp` file does not have the `user_type` property set. If this property is omitted, the user is assumed to be internal. When this property is set, it indicates that an external user is logging in. This property can be set to any value other than `INTERNAL`.

Customization of the Login Page

The `logon.jsp` file can be customized; that is, the `logon.jsp` file can be replaced by a custom `logon.jsp` file, for a number of reasons.

The reasons the file can be replaced include the following: reasons.

An external user client application is being developed
If an external user client application is being developed, a new `logon.jsp` file needs to be created, as the user type needs to be set to indicate that an external user is logging in. For more information, see "Creating an External User Client Login Page" on page 42.

Automatic login is needed
Some external user client applications require no user authentication and hence a `username` and `password` need not be requested, that is, if an external public access application. It is not possible to disable authentication, so the best way to achieve this requirement is to write an automatic login script. This procedure is done by customizing the `logon.jsp` file for the external public access application. For more information, see "Creating an External User Client Automatic Login Page" on page 42.

Different styling is required
The section on Login Pages in the *Cúram Web Client Reference Manual* for mower information on styling for the `logon.jsp` file.

A requirement exists for user names to contain extended characters (valid only for Oracle WebLogic Server)
Web Logic Server provides a proprietary attribute, `j_character_encoding`, which must be added to the `logon.jsp` file. For more information, see "Enabling Usernames With Extended Characters for WebLogic Server" on page 30.

Cúram JAAS Login Module

Authentication is performed by a Java Authentication and Authorization Service (JAAS) login module. It is configured in the application server and is started automatically by the application server as part of the authentication process for any access to the Cúram application. The advantage to this approach is that the default authentication mechanism can be used with, or replaced by, a custom approach, without affecting the Cúram application.

As mentioned earlier, the Cúram JAAS login module can be configured to operate in three modes. For more information on the configuration of the login modules and any application server-specific behavior, see the section on Application Server Configuration within the *Cúram Server Deployment Guide* for the application server that is being used.

Project specific requirements might mean that more than one login module is needed, for example, a user might be required to enter more than the `username` and `password` for verification purposes. It is possible to configure multiple login modules in the application server. Each login module is run in the order as determined by the settings in the application server.

For more information on these settings, see the WebSphere or WebLogic Server documentation.

After the user is authenticated successfully by all login modules that require successful authentication of the user (this login is configurable in the application server), the user is considered authenticated by the application.

Password Management

The passwords for all Cúram internal and external users are stored in their digest format on the `Cúram Users` and `ExternalUsers` database tables. When the Cúram Java Authentication and Authorization Service (JAAS) login module receives the `password`, it is digested before it is sent to the login bean for comparison.

Digesting is a one-way process to ensure the security of the password. The `password` stored for the user on the database uses the same digest algorithm, subject to your encryption settings, ensuring the encrypted passwords can be compared successfully to each other, but remain secure.

Users who are managed externally, for example, through Lightweight Directory Access Protocol (LDAP) with Cúram identity-only configured, are not subject to the process described previously. When a user is being authenticated against a third-party party system (for example, LDAP or a Single sign-on (SSO) Server), where a need exists for the Cúram application to pass the user-entered credentials to the third-party system, the custom implementation of `curam.util.security.PublicAccessUser` can be used. This process allows access to the credentials with a plain-text `password`.

Default Configuration for WebLogic Server

The Cúram Java Authentication and Authorization Service (JAAS) login module is configured as an authentication provider in WebLogic Server. The Cúram authentication provider is the only provider configured by the configuration scripts provided for WebLogic Server. Since it is the only configured authentication provider, the Cúram authentication provider is responsible for authenticating and verifying the user.

As mentioned previously, it is possible there might be more than one authentication provider configured in WebLogic Server. In this case, the Cúram authentication provider might not be responsible for authenticating and verifying the user. For more information, see "Single Sign On for WebLogic Server" on page 27.

Default Configuration for WebSphere

The Cúram Java Authentication and Authorization Service (JAAS) login module is configured as a system login module in WebSphere. The default, scripted security configuration within WebSphere involves the default file-based user registry and the Cúram system login module.

The user registry in WebSphere is the default authentication mechanism and can be configured to be:

- A custom user registry
- A Lightweight Directory Access Protocol (LDAP) directory server
- The Local operating system (OS)
- The WebSphere file-based repository

Multiple system login configurations exist for WebSphere. The Cúram system login module is configured for the `DEFAULT`, `WEB_INBOUND`, and `RMI_INBOUND` configurations. The same login module is used for all three configurations. WebSphere automatically starts the login modules configured as system login modules under certain circumstances:

- `DEFAULT`

  The login modules that are specified for the `DEFAULT` configuration are started for authentication of web services and JMS invocations. They also are started during the startup phase of WebSphere
- `WEB_INBOUND`

  The login modules that are specified for the `WEB_INBOUND` configuration are used for authentication of web requests
- `RMI_INBOUND`

  The login modules that are specified for the `RMI_INBOUND` configuration are used for authentication of Java clients.

The Cúram JAAS login module exists as a login module within a chain of login modules that are set up in WebSphere. It is expected that at least one of these login modules be responsible for adding credentials for the user. By default, the Cúram login module adds credentials for an authenticated user. As a result of this process, the configured WebSphere user registry that is handled by a subsequent login module does not add credentials.

Therefore, it is not necessary to define Cúram users within the WebSphere user registry. This behavior is configurable by using the `curam.security.user.registry.enabled` property set in the `AppServer.properties` file. For more information on setting this property, see *Cúram Deployment Guide for WebSphere Application Server* or *Cúram Deployment Guide for WebSphere Application Server on z/OS*.

This figure illustrates the default authentication flow for WebSphere.

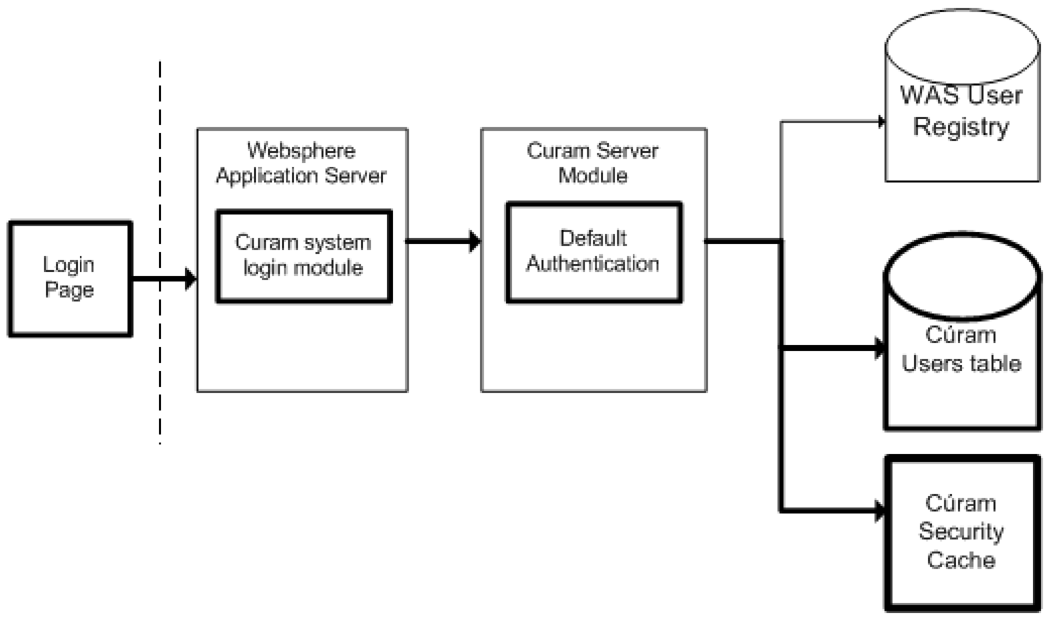

Figure 3. Default authentication flow for WebSphere

This figure illustrates the authentication flow for WebSphere where its user registry is also queried, that is, where the `curam.security.user.registry.enabled` property is set to `true`.

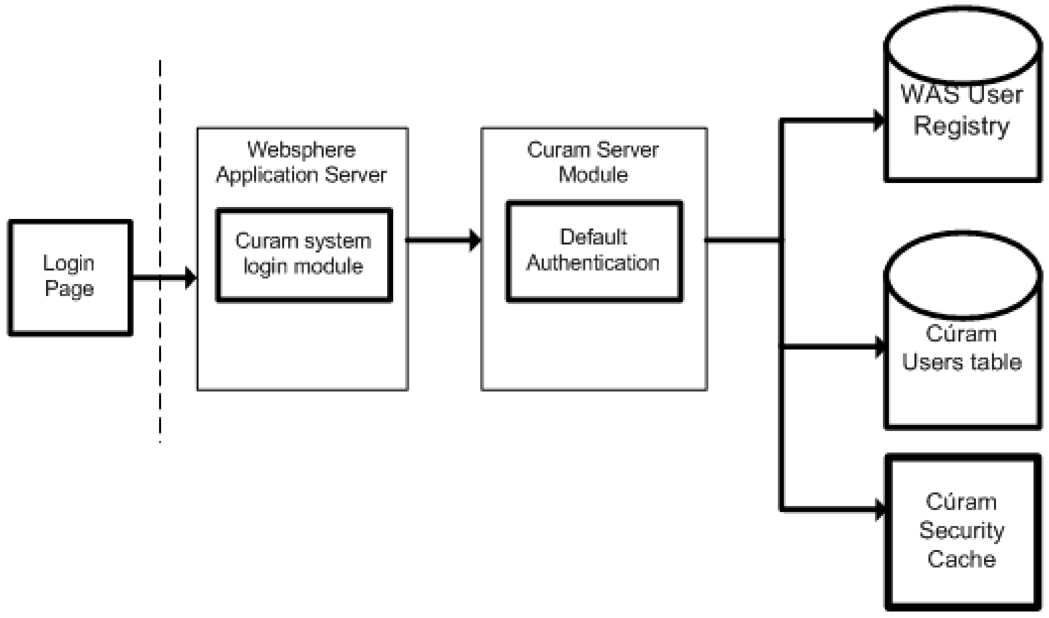

Figure 4. Authentication Flow for WebSphere with User Registry Enabled

As part of the security configuration, certain users exist that are excluded from authentication and for these users the configured user registry is queried. This list of users is configured automatically to be the WebSphere security user, as specified by the `security.username` property in `AppServer.properties` and the database user, as specified by the `curam.db.username` property in `Bootstrap.properties` . These two users are classified administrative users and not application users. It is possible to extend this list of excluded users manually. For more information, see the *Cúram Deployment Guide for WebSphere Application Server* and *Cúram Deployment Guide for WebSphere Application Server on z/OS*.

Warning: The `security.username` and `curam.db.username` users are automatically added to the WebSphere file-based user repository by the provided configuration scripts. If the configured WebSphere user registry is not the default, these users must exist in the alternate WebSphere user registry.

Customization of the JAAS Login Module

It is possible that the Cúram Java Authentication and Authorization Service (JAAS) login module might not support the authentication requirements for a particular custom solution. We strongly recommend that when users develop a custom login module, that the Cúram JAAS login module needs to be left in place and used with identity only authentication enabled. However, if deemed necessary, the Cúram JAAS login module can be removed and replaced by a custom solution. If this is the case, Support must be consulted.

Warning: While it is possible to remove the Cúram JAAS login module completely, it needs to be noted that users must still exist in the Cúram Users database table for authorization reasons.

The Cúram JAAS login module adds new users to the Cúram Security Cache automatically, and when this Cúram JAAS login module is replaced by a custom JAAS login module, this function no longer is present. If a custom JAAS login module is replacing the Cúram JAAS login module completely, it is the responsibility of the custom JAAS login module to ensure that an update of the Security Cache is triggered when a new user is added to the database.

Verification Process for Authentication

The type of verifications that are performed depends on the authentication mode that is being used. The following list shows authentication modes and configurations with full details on the verifications completed for each authentication mode.

Authentication Overview

In Cúram, authentication is the process of determining if a user is who they say they are. Authentication is needed where a user must be verified in order to access a secure resource on a system.

Form-based authentication is where a user is presented with a form allowing them to enter username and password credentials. These credentials are compared against the credentials stored on the system for this username, if they match the user is considered an authenticated user for the system. For security reasons the password for authenticating a user is stored on the system in a digested form.

The Cúramweb client is configured to support form-based authentication, which means that before a user can access any of the web client content, they will be redirected to a login form to authenticate.

The authentication process involves the verification of the username and password, and this is performed by default by a JAAS (Java Authentication and Authorization Service) login module. HTTPS/SSL is turned on by default in the web client ensuring the form-based login authentication mode is secure.

Default Authentication:

Default authentication is part of the initial configuration and this mode of authentication involves verifying the `username` and `password` specified during login against the `Cúram Users` database table. All login information in this case is maintained by the Cúram application.

Default Verification Process:

Several verifications are required by the Cúram login module during default authentication. These verifications include queries that include the user name, password, and account information.

The verifications included during the default authentication are:
- `username` and `password`.
- Account and password expiry
- User name synchronization with security cache

- Break-in detection, for example, upper limit on password entry attempts, incorrect user names, password change failures
- Day and time access restrictions - day of the week and time range within the day

The authentication and authorization of user names is case sensitive by default. However, it is possible to disable case-sensitive authentication. If duplicate case insensitive user names exist (for example, caseworker, CaseWorker), authentication fails due to an ambiguous user name. For more information, see "Changing the Case-Sensitivity of the Username" on page 30.

Authentication Attempts:

Authentication failures are not reported directly to a client as this reporting would provide extra information to an intruder who is attempting to break into the system. For example, reporting an incorrect password would indicate that the user name is valid.

All authentication attempts (both success and failure) instead are logged in a database table called the `AuthenticationLog`.For more information, see "Analyzing the AuthorisationLog Database Table" on page 36.

Customization of Default Authentication:

The default implementation can be customized to use a mutable `login ID` instead of the Cúram `username` and the ability to add extra verifications is available by implementing the custom authenticator.

For more information, see "Custom Verifications" on page 14.

Identity Only Authentication:

Identity only verification means that the authentication mechanism only ensures that the user name for the user who is logging in exists on the `Cúram Users` database table. Full authentication must be completed by an alternative mechanism to be configured in the application server.

Authentication can be configured to perform identity-only verification, in place of the default verifications listed in "Default Verification Process" on page 11.

An example of an alternative mechanism is a Lightweight Directory Access Protocol (LDAP) directory server, which is supported as an authentication mechanism by both the WebSphere and WebLogic Server application servers. Another alternative is to use a Single Sign-On (SSO) Solution for authentication, or to implement a custom login module. For custom application server solutions, the IBM or Oracle documentation needs to be consulted.

With identity-only authentication (as for default authentication), entries are added to the `AuthenticationLog` database table at the end of the authentication process.

For a successful login the following status is used:
- `AUTHONLY`

For a failure scenario, the following status is used:
- `BADUSER`

This scenario is the only possible failure scenario where a user does not exist.

The `loginFailures` and `lastLogin` fields of the `AuthenticationLog` are not set. This condition is true even if customized verifications are implemented.

When the password expiry information for a user is set (on the `Cúram Users` database table), the password expiry warning is displayed if it is about to expire. With identity-only authentication, this warning is misleading. It is recommended that any fields that relate to the authentication verifications, such as password expiry or account enabled, are not used if identity-only authentication is enabled.

When identity-only authentication is enabled, security is not used for authentication but is still used for authorization purposes. As a result of this requirement, all users who require access to the application needs to still exist in the `Cúram Users` database table, and in the alternative authentication mechanism, for example, Lightweight Directory Access Protocol (LDAP).

Note: Two users must exist in both locations, that is, the `SYSTEM` user and the `DBTOJMS` user. For more information, see "Security for Alternative Clients" on page 22.

For more information on how to configure identity only for an application server, see "Configuring Identity Only Authentication" on page 31.

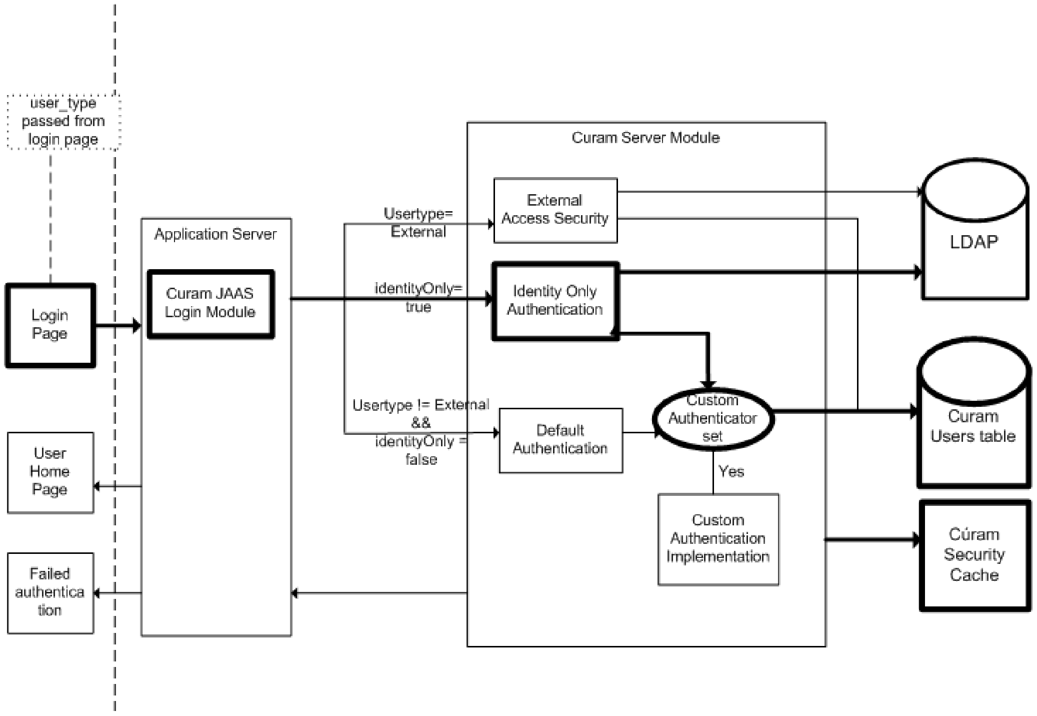

Figure 5. Identity Only Authentication

Customization of Identity Only Authentication:

The identity-only implementation cannot be customized, but extra verifications can be added by implementing the custom authenticator.

For more information, see "Custom Verifications" on page 14.

External Access Security Authentication:

The architecture allows a developer to implement their own custom authentication solution for external users by providing a hook into the existing authentication and authorization infrastructure of the Structured Query Language for Java (SQLJ).

To hook the custom solution into the application, the `curam.util.security.PublicAccessUser` class must be extended, which requires implementing the `curam.util.security.ExternalAccessSecurity` interface. This class is used during the authentication and authorization process to determine required information that is related to the External User.

For more information, see "Customizing External User Applications" on page 41.

Custom Verifications:

Support is provided for adding custom verifications to the authentication process. For example, a user might be required to answer a security question that must then be verified. The custom code, if implemented, is started after the relevant Cúram verifications or identity assertion, and only if they were successful.

After the custom verifications are started, the authentication process will update the relevant fields on the `Users` database table.

For more information, see "Adding Custom Verifications to the Authentication Process" on page 30.

Authorization Overview

In Cúram, authorization is the process of granting or refusing a user access to functional elements of an application.

The functional element can be anything to which a unique identifier can be attached, such as:

- a server process call,
- an element of the application that requires security checking, for example, a series of registered welfare products.

Access to the functional element is controlled by a Security Identifier (SID) that forms part of the Cúram authorization data. This data is linked to a user and can be configured through the Cúram Administration screens or through the Data Manager. For more information, see the *Cúram Server Developer's Guide*.

The security data that is created for authorization is central to the processing performed during every client-server call, and it is important that access is optimized for performance reasons. The Cúram Security Cache is responsible for caching authorization data for a user. For more information, see "Cúram Security Cache" on page 21.

The following topics describe the relationship for these authorization concepts and how authorization works within Cúram.

Users, Roles and Groups

The security information associated with an application must first be organized into security profiles before it can be utilized in a runtime environment. A security profile consists of a security role, one or more security groups and the associations between security identifiers (SIDs) and securable elements of an application.

Every authorized user is assigned a security role during security configuration and these roles are associated with a number of security groups. Each security group is associated with a number of security identifiers. The security identifier represents the securable elements of Cúram, for example., a method or a field. The role, groups and identifier information is stored on the database in a number of tables and is configured using the application Data Manager or the Cúram Administration screens.

This data structure makes it possible to authorize every user against any secured element of an application. This is a powerful and flexible method of providing authorization to Cúram users.

There is a minimum set of SIDs required for a user to operate the Cúram Platform application. These SIDs are associated to the out-of-the-box BASESECURITYGROUP group. The `EJBServer/components/core/data/initial/handcraftedscripts/Supergroup.sql` file should be consulted to identify the list of these SIDs. This file is responsible for linking the SIDs to the BASESECURITYGROUP out-of-the-box.

A simple way to ensure that all users have the privileges from this set of SIDs is to create a single security group for them and then associate that security group with every security role in the system.

Security Identifiers (SIDs)

Every secured element in Cúram is given a security identifier (SID) that is unique across the entire application.

The authorization process is built into the infrastructure and once the securable elements have been identified, the rest is handled by code generators, scripts and the Cúram Administration screens. The analysis of what elements must be securable is a manual process that must be done by the developer or security administrator. This section outlines the infrastructure available to set up authorization.

The first type of authorization to consider is that of the process method(facade) also known as *function-level security* . In the Cúram model, a developer may choose if security is switched on or off at the process method level. The option applies only to Business Process Objects (BPOs) since they encapsulate the calls exposed to the client. Entity object methods are not included in the authorization process.

There are a number of types of SIDs and these include:
- Function Identifiers (FIDs)
- Field Level Security Identifiers
- User defined SID types.

Function Identifiers (FIDs)

Function identifiers (FIDs) are a specialized type of security identifier (SID) where the type is set to FUNCTION. When a method is made publicly accessible (by setting the stereotype as facade in the model), a FID is generated for that method and security is automatically turned on.

It is possible to turn off security for a process method at design time. For more information, see "Switching Security off for a Process Method" on page 35.

Adding an FID

To add an FID, do the following steps:
1. Log on as the `sysadmin` user and click **System Configurations**.
2. In the Shortcuts panel, click **Security** > **Identifiers**.
3. In the actions menu, click **New Function Identifier** and enter the details for the FID.
4. In the actions menu, click **Publish**.
5. In the Shortcuts panel, click **Security** > **Groups**.
6. Click a group to add the FID to, and then click **Add Identifiers**.
7. From the list of alphabetically ordered identifiers that is displayed, select the identifier that your created and click **Save**.
8. Click **Publish**.

Field Level Security Identifiers

The Field Level SID allows authorization to be applied to specific fields on a publicly accessible method. At runtime, if a user does not have access rights to view the field to be displayed, the contents of the field are displayed as a number of asterisks (***). For more information on Field Level SIDs , the *Cúram Modeling Reference Guide* should be consulted.

User Defined SIDs

In the previous sections, we have described

FIDs; An automatically generated SID of type function.

Field Level SID;
Security applied to specific fields on a method.

There is also the concept of a user defined SID. The authorization process is sufficiently flexible to accommodate any securable element of an Cúram application. The developer can effectively customize the authorization process by defining new *types* of SIDs. The new types represent a conceptual element requiring security. The following server interface method enables authorization to be invoked directly on these new user defined SID types.

```
curam.util.security.Authorisation.isSIDAuthorised()
```

Out-of-the-box, the LOCATION and PRODUCT SIDs are SIDs of this type. Using the above method there is effectively no limit to the SID types that can be defined. "Authorizing New SID Types" on page 36 should be consulted for further details.

Runtime Authorization

The Cúram infrastructure performs authorization checks from both the web client and server side.

Client Authorization Checks

Before a user can access a method or field, the web client performs authorization checks before the page is initially loaded. If the user does not have access, the client authorization check fails, and the server is not invoked. This check is configurable in the `curam-config.xml` by setting the SECURITY_CHECK_ON_PAGE_LOAD property. Section 3.12.13 General Configuration in the *Cúram Web Client Reference Manual* should be consulted for further details on this.

By default any such web client authorization failures are not recorded. This behavior is configurable. "Controlling the Logging of Authorization Failures for the Client" on page 36 should be consulted for further details.

Server Authorization Checks

To cater for other access to Cúram, and where the web client authorization check is disabled, there is a second level authorization check made by the server. This server side check will always log authorization failures, and the client property does not affect this logging.

The log of all authorization failures is stored on the database to allow these failures to be audited at a later stage. The AuthorisationLog table contains the User Name and Security Identifier for the failed authorization, as well as a timestamp indicating when the failure occurred. "Analyzing the AuthorisationLog Database Table" on page 36 should be consulted for further details on the AuthorisationLog table.

Cryptography in Cúram

In Cúram, cryptography refers broadly to ciphers and digests, two types of functionality that are related to keeping your Cúram systems safe and secure.

You can use ciphers and digests as follows in Cúram:

- ciphers - for two-way encryption of passwords that are used at various processing points
- digests - for one-way hashing (or digesting) of passwords; for example, used at login

You can select the values for configuring cryptographic behavior with the `CryptoConfig.properties` property file to provide you with the most control and security possible for your Cúram installation. This flexibility provides the capability to adjust to changing security standards. For more information about configuring and customiziing cryptography, see "Customizing Cryptography" on page 37.

If you are migrating for the first time to a level of Cúram that has this level of cryptographic support, which was introduced in version 6.0.5.0, it is recommended that you upgrade system (new cipher) and user (new digest) passwords from the default values to improve your security.

Supported cryptographic configurations are:

1. AES: 128, 192, 256 (FIPS 140-2 and SP800-131a compliant);
2. Two-key Triple DES - DESede: 112 (FIPS 140-2 compliant);
3. Three-key Triple DES - DESede: 168 (FIPS 140-2 and SP800-131a compliant);
4. No cryptography configuration, which is configured by removing the `CryptoConfig.properties` file, in which case Cúram reverts to its previous default crypto settings.

In the environment where Cúram runs, the application server, database, and other software, such as web server or LDAP software, has its own cryptographic support and you can refer to the relevant vendor's documentation.

Ciphering

Ciphering refers to the process of encrypting passwords, which are listed in “Cipher-Encrypted Passwords” on page 20. That is, this is a two-way process representing decrypt-able values. There are about a dozen of these encrypted passwords in various property files in Cúram and encrypting them helps keep them secure and they are are decrypted at the necessary points for usage; e.g. connecting to your database system.

Digesting

Digesting refers to the one-way process of handling passwords that do not require decrypting, but is used for storing passwords for later comparison; e.g. Cúram user logins. That is, this is a one-way process representing non-decryptable values.

Cryptography Properties

The Cúram `CryptoConfig.properties` file contains settings for cipher and digest cryptography. Therefore, this file and all the files it refers to (i.e., keystore and salt) should be considered critical items to the security of your system and should be provided with adequate access controls (e.g., file permissions) and specifically modified and segregated when used for production systems. That is, if the details of these files were to become widely known, while not necessarily a security risk themselves, would remove a level of protection that might necessitate a disruptive crypto change (see “Cipher Customization” on page 37 and “Digest Customization” on page 39).

Related topics:
- “Cúram Cipher Settings”
- “Cúram Digest Settings” on page 19

Cúram Cipher Settings

Various passwords within Cúram property files and configurations are stored in an encrypted format out-of-the-box (OOTB).

The Cúram crypto configuration will work for you out-of-the box, but it is recommended you modify these settings with respect to your local security requirements. For instance, the OOTB settings may be adequate in development, but for production environments it is strongly recommended that you modify them (e.g. by changing the cipher secret key).

The cipher settings are stored in the `CryptoConfig.properties` file. The properties and their values are as follows:
- `curam.security.crypto.cipher.algorithm`
  - **Valid values:** In JCE documentation, for example: http://docs.oracle.com/javase/6/docs/technotes/guides/security/StandardNames.html#Cipher. The supported ciphers are AES and the various forms of Triple DES.
  - **Default:** AES (FIPS 140-2 and SP800-131a compliant)
- `curam.security.crypto.superseded.cipher.algorithm`
  - **Valid values:** See `curam.security.crypto.cipher.algorithm`
  - **Default:** None

- **Purpose:** Provides for flexibility to support an upgrade/migration period for Cúram user passwords with custom code (e.g. a batch program) via the `curam.util.security.EncryptionUtil.decryptSupersededPassword()` API. The use of an upgrade/migration period is explained in more detail in "How to Utilize the Superseded Digest Settings for a Period of Migration" on page 40.

- `curam.security.crypto.cipher.keystore.location`
  - **Valid values:** Path to keystore file containing secret key. This can be an absolute path specification or relative to the classpath (e.g. `CuramSample.keystore`).
  - **Default:** None
- `curam.security.crypto.cipher.keystore.storepass`
  - **Valid values:** As per the JDK **`keytool`** command.
  - **Default:** password
  - **Purpose:** Specify the password used to access the keystore.
- `curam.security.crypto.cipher.provider.class`
  - **Valid values:** Fully-qualified name of a JCE cryptography provider class.
  - **Default:** blank
  - **Purpose:** Optional way to enable the use of an alternate standards-compliant provider.

This ciphering functionality applies to the properties as described in "Cipher-Encrypted Passwords" on page 20.

These Cúram cryptographic settings are enabled by default OOTB and represents changes that existing Cúram installations must address as documented in the *Cúram Upgrade Guide*.

Cúram Digest Settings

Cúram users, internal and external, when not invoked with identity-only, are authenticated using form-based login and the password entered in the form is digested and compared to the digest value stored in the database for the user.

Note: This processing does not apply to users authenticated in third party systems like LDAP.

The Cúram crypto configuration will work for you out-of-the box, but it is recommended you modify these settings with respect to your local security requirements. For instance, the OOTB settings may be adequate in development, but for production environments it is strongly recommended that you modify them (e.g. digest salt encrypted value).

The digest settings are stored in the `CryptoConfig.properties` file. The properties and their values are as follows:

- `curam.security.crypto.digest.algorithm`
  - **Valid values:** In JCE documentation, for instance: http://docs.oracle.com/javase/6/docs/technotes/guides/security/StandardNames.html#MessageDigest. The supported digests are the SHA variants (1, 256, etc.) and MD5.
  - **Default:** SHA-256 (FIPS 140-2 and SP800-131a compliant)
  - **Purpose:** Specification of the digest algorithm.
- `curam.security.crypto.digest.salt.location`

  - **Valid values:** A path identifying the file containing the encrypted secret digest salt.
  - **Default:** None
  - **Purpose:** An optional file to specify the salt (encrypted) for digesting.
- `curam.security.crypto.digest.iterations`
  - **Valid values:** 0 or a positive integer.
  - **Default:** 0
  - **Purpose:** Typically, higher values give better security, but at the cost of processing (e.g. at login time).

There are a set of corresponding "superseded" properties to allow for flexibility when migrating from one set of digest settings or standards to another. The following have a similar function to their counterparts above, but are used by the Cúram encryption functionality to support both old and new settings for a time of migration:

- `curam.security.crypto.superseded.digest.algorithm`
- `curam.security.crypto.superseded.digest.salt.location`
- `curam.security.crypto.superseded.digest.iterations`

The usage and behavior of the superseded properties are controlled by the `curam.security.convertsupersededpassworddigests.enabled` property as managed by the Properties Administration user interface. See "How to Utilize the Superseded Digest Settings for a Period of Migration" on page 40 for more information on using the superseded properties.

Cipher-Encrypted Passwords

The following passwords are cipher-encrypted in Cúram:

- `Bootstrap.properties`:
  - `curam.db.password` - database password
  - `curam.searchserver.sync.password` - see *Cúram Generic Search Server* for more information
- `AppServer.properties`: (typically this property file is used for configuring test servers and is not appropriate for production systems)
  - `security.password` - application server administration console password
  - `curam.security.credentials.async.password` - replacing the `runas.password` property
- `Application.prx` - individual property descriptions are as documented with the properties in the Curam Property Administration user interface:
  - `curam.security.credentials.dbtojms.password` - (in conjunction with `curam.security.credentials.dbtojms.username`), which replaces the curam.omega3.DBtoJMSCredentialsIntf interface APIs previously used to provide custom credentials for DB-TO-JMS
  - `curam.security.credentials.ws.password` (in conjunction with `curam.security.credentials.ws.username`), which replaces the build-time default web services default credential settings.
  - `curam.meeting.request.reply.password` - (an SMTP password)
  - `curam.ldap.password`
  - `curam.citizenworkspace.password.protection.key`
- `BIBootstrap.properties` - BIRT users only; see the *Cúram Business Intelligence BIRT Developer Guide*:

  - `curamsource.db.password`
  - `central.db.password`
  - `centraldm.db.password`
- Web Services - See the *Cúram Web Services Guide*:
  - `ws_inbound.xml` - `<ws_service_password>`
  - `services.xml` - `<parameter name="jndiPassword">`
- CTM - Cúram Transport Manager:
  - The Password column of the TargetSystemService table contains an encrypted password

Security Data Caching

An overview of the Cúram Security Cache, which stores all authorization data for a user. Details on the WebSphere cache and how this affects the authentication of a user at login are also included.

Cúram Security Cache

Security information from the database tables supporting the profiles mentioned in "Users, Roles and Groups" on page 14 is cached by the infrastructure. This is done to optimize the search and retrieval of data during the authorization process.

To optimize performance, the cache is loaded on demand as security authorization requests come into the application and is a shared resource. For application code, the cache is a protected resource and cannot be accessed directly. It is accessible, for queries only, through the authorization interface (`curam.util.security.Authorisation`) which allows a developer to implement a customized authorization procedure. "Authorizing New SID Types" on page 36 should be referenced for further details on this.

When the curam.security.casesensitive property is set to false the security cache will store all usernames in upper case and all queries to the cache will automatically change the specified username into the upper case equivalent. It is also worth noting that the existence of duplicate case insensitive usernames will cause a fatal error during the initialization of the security cache. "Changing the Case-Sensitivity of the Username" on page 30 should be consulted for further details on this.

Cache Refresh

As security data is so important to the operation of Cúram , the cache must be refreshed whenever any changes have been made to security related database tables. The refreshing of the Cúram Security Cache is an asynchronous process.

Cache Refresh Failure

The refreshing of the Cúram Security Cache is triggered by either an application reboot, or by the system administrator (sysadmin) via the Cúram Administration screens, therefore, the administrator receives no feedback if the cache reload fails. Having to check the system logs or manually verify the application following a refresh to verify its success can be cumbersome. It is therefore recommended that the optional callback interface for providing feedback in the event of a cache reload failure be implemented. "Adding the Cache Refresh Failure Callback Interface" on page 31 should be consulted for further details.

WebSphere Caching Behavior

WebSphere caches user information and credentials in its own security cache. The Cúram login module will not be invoked while a user entry is valid in this cache. The default invalidation time for this security cache is ten minutes, where the user has been inactive for ten minutes.

For example, the first time a user logs into the application from the web client they will be requested for their username and password. The Cúram login module will be invoked, and will authenticate the information specified. If the same user opens a second new web browser and attempts to access the application, they will again be requested for their username and password. When WebSphere receives this information it will query the security cache to determine if the username and password are already in the cache. If they are, and the password matches, WebSphere will not query the login modules.

The impact of this behavior is that any modifications to a user's account restrictions or password will not take effect until the user has been invalidated from the WebSphere security cache.

For more information see the appropriate *WebSphere Application Server Information Center*.

Security for Alternative Clients

Certain processes cannot be associated with a specific logged-in user. These include alternative clients, for example, non-web processes such as batch processing, web services, and deferred processing. As any process that interacts with a Cúram application must be authenticated, a valid user must exist for each of these processes. These topics provide details on the users that must exist on the Cúram Users table and details on the processes that depend on these users.

Mandatory Cúram Users

A number of users must always exist in the Cúram Users database table. These users are necessary for application processes such as deferred processing and workflow. If these users do not exist, then authentication will fail and subsequently these processes will fail.

The usernames and passwords for each of the processed below are the default out-of-the-box credentials and it is recommended that these credentials be changed for security reasons.

These users include:

- SYSTEM

  The SYSTEM user is the user under which JMS messages are executed. This user must exist and the username is case sensitive. “JMS Messaging” on page 23 should be referenced for further details.
- DBTOJMS

  The DBTOJMS user is the default user under which the Database to JMS (DBToJMS) trigger for batch processing is executed. This user must exist and the username is case sensitive. “Batch Processing” on page 23 should be referenced for further details.
- WEBSVCS

The WEBSVCS user is the default user under web services are executed. This user must exist and the username is case sensitive. "Web Services" should be referenced for further details.

Web Services

For Apache Axis2 (the recommended implementation for web services) there are default credentials for authentication. A user has the ability to change these credentials at a global level or per service if required. To ensure that web services are not vulnerable to a security breach this default user is not authorized to access web services by default. For authorization, a web service must be associated with a security group and in turn a security role that is linked to the user (e.g. WEBSVCS) in order to access it. Ensuring the user is authorized is a manual process. Please see the *Customizing Receiver Runtime Functionality* section in the *Cúram Web Services Guide* for further details on web services and also the chapter on Authorization in this book.

For Apache Axis 1.4, i.e. legacy web services, once a process is modeled as a web service, this web service will automatically be logged into the application using default credentials. This default user is set up for authorization automatically, i.e. the user will have access to the web service created. Therefore caution is advised when making a class visible as a web service. Please see the *Legacy Inbound Web Services* section within the *Cúram Web Services Guide*.

There are a number of other topics related to the security of web services - for example, encrypting data - using Rampart. The *Cúram Web Services Guide* should be consulted for further details on these.

Batch Processing

Since the Batch Launcher does not require the application server to be running, it does not perform any application level authentication or authorization. It must only authenticate against the database. The same credentials as used by the application server (located in `%SERVER_DIR%/project/properties/Bootstrap.properties`) are used by the Batch Launcher to connect to the database and run batch programs.

The Batch Launcher or batch programs can optionally trigger the application server to begin a DB-to-JMS transfer. This involves logging in and invoking a method on the server, which in turn requires a valid username and password. By default the DB-to-JMS transfer operation uses default credentials; therefore, the DBTOJMS account must exist on the Cúram Users table and must be enabled and assigned the role 'SYSTEMROLE' to allow authorization. The locale DB-to-JMS transfer is the default locale for this user as specified in field 'defaultLocale' on the Users table.

The Security Considerations section in the *Cúram Batch Processing Guide* guide should be consulted for further details on changing the user for the DB-to-JMS transfer.

The property batch.username can be used to specify the user name for the operations run by the Batch Launcher. This is set using the -D parameter. For example: build runbatch -Dbatch.username=admin

JMS Messaging

JMS messages are used for communication purposes by deferred processes and Workflow. Since JMS messages are triggered by the application server and need to interact with the Cúram application, valid Cúram credentials must exist. The SYSTEM user account must exist on the Cúram Users table and must be enabled and assigned the role 'SYSTEMROLE' to ensure authorization. The locale for JMS messages is the default locale for this user as specified in field 'defaultLocale' on the Users table.

It is possible to change the SYSTEM username during or after the deployment of the application. For more information the *Cúram Server Deployment Guide* for the relevant application server should be consulted.

Deferred Processing

A deferred process in Cúram is a business method that is invoked asynchronously. As deferred processes interact with the application, valid Cúram credentials must exist. The SYSTEM user account must exist on the Cúram Users table and must be enabled and assigned the role 'SYSTEMROLE' to ensure authorization. The locale for deferred processes is the default locale for this user as specified in field 'defaultLocale' on the Users table. In the case of offline unit-testing of deferred processes, the username is blank and the effective locale is the default locale for the Cúram server.

External User Applications

Typically, there are users outside the organization with limited access who needs to securely access parts of the Cúram application. These users are considered external users and authentication for these users is completely customizable through the use of the External Access Security hook point provided. As external users are processed differently to internal users, a specific web application is required for external users.

The default Cúram application is enabled for internal users. Internal users are users that exist on the Cúram Users database table. A typical internal user would be a case worker who creates and manages claims for participants and has full access to the application. The infrastructure provides functionality for authenticating and authorizing these internal users.

External User Applications

When developing an application for an external user, the following must be implemented:

- An external user client application, i.e., a separate EAR file containing the web client application.
- A custom `logon.jsp` , where the external application must pass in a parameter user_type indicating an external user is logging in.
- A custom class that extends `curam.util.security.PublicAccessUser`, which requires implementing the `curam.util.security.ExternalAccessSecurity` interface, must be provided. This abstract class contains methods responsible for the authentication and authorization of an external user.

As well as there being internal and external user types. There can also be different types of external users. For example, there may be an external user of type 'PUBLIC' who could have limited access to an external application. There could be another external user of type 'PROVIDER' who is a registered external user. The ability to have different types of external users provides more flexibility within an external application, allowing finer grained control over authentication of the external user based on the external user type.

User Scope

There are two different types, or scopes, of users within the Cúram application: internal and external. The type of a user is determined in one of the following ways:

- By the Cúram Security Cache;

  If the user exists in the Cúram Security Cache, the type is assumed to be in internal. If the user does not exist in the cache, the type is assumed to be external. In this case, (which is the default behavior) all usernames, internal and external, must be unique.
- By the `UserScope` custom interface;

  If the `UserScope` custom interface is implemented. This custom interface, takes precedence over the check for a user in the Cúram Security Cache to determine the user type. Consult "Determining if a User is Internal or External using the UserScope Interface" on page 49 for further details.

When the type of a user is external the implementation of the `curam.util.security.ExternalAccessSecurity.getSecurityRole()` method will be used to determine the user role instead of the internal security roles. "Authorizing an External User" on page 46 should be consulted for further details on this method.

To support alternative methods for determining if a user is internal or external the custom interface, `UserScope` , is available. Consult "Determining if a User is Internal or External using the UserScope Interface" on page 49 for more details.

Deployment of an External Application

When deploying an application to an application server, the security configuration for the application server is applicable to all Cúram applications deployed to that application server instance. Therefore, care must be taken when considering the deployment architecture for more than one application. This is important when deciding if an internal and external application will be deployed to the same application server instance.

An example of some considerations to think about are:

- Is identity only being used for internal users?
- Is an alternative authentication mechanism used , e.g., LDAP;
- Will both internal and external users be authenticated by LDAP?

The answers to the considerations above will affect the setting of the application server properties (i.e. properties specified in the `AppServer.properties` file), that affect the behavior of the Cúram JAAS login module. These considerations will also drive the implementation of the `curam.util.security.PublicAccessUser` class and `curam.util.security.ExternalAccessSecurity` interface for external users.

The application server properties in the Cúram JAAS login module allow for finer grained control over the authentication of user types. External users and internal users can be authenticated differently, as can different types of external users, in a situation where the internal and external applications are deployed to the same application server. These properties include the following:

- curam.security.user.registry.disabled.types ;

  Set this property to a comma separated list of user types for which the application server user registry *will not* be queried, i.e. the implementation within the `PublicAccessUser.authenticate()` method is responsible for authenticating the external user of this type. For example, LDAP could be configured to be the user registry.

- curam.security.user.registry.enabled.types.

  Set this property to a comma separated list of user types for which the user registry *will* be queried, i.e., the implementation within the `PublicAccessUser.authenticate()` method does not have to fully authenticate the user. The user registry will be responsible for authenticating this type of external user. For example, LDAP could be configured as the user registry, and in this case, LDAP could be responsible for the authentication of these external user types.

These properties are dependent on the implementation of the `curam.util.security.PublicAccessUser` class and `ExternalAccessSecurity` interface.

Consider the following example project requirements:
- An internal user must authenticate with LDAP.
- An external user of type 'EXT_PUBLIC' must authenticate with Cúram and not LDAP;
- An external user of type, 'EXTERNAL' must authenticate with LDAP only and not Cúram.
- Both the internal and external applications are deployed to the same application server instance.

The following settings could cater for the example above:
- curam.security.check.identity.only set to `true` ;
- curam.security.user.registry.disabled.types=EXT_PUBLIC.

As well as the properties being set, the `PublicAccessUser` extension (and `curam.util.security.ExternalAccessSecurity` implementation) must have the logic to cater for the different types of external users and how they will be authenticated.

Using Single Sign On

Single sign-on (SSO) allows users to access multiple secure applications by authenticating only once. Single sign-on is supported for the Cúram supported application servers, by allowing alternative mechanisms to be used alongside the Cúram login module. Cúram application server properties allow use of an SSO solution.

The number of applications in an enterprise often results in an increase in the number of user names and passwords in use, resulting in poor user experience and extra maintenance costs. Multiple user names and passwords also compromise security as users either choose very simple passwords or write down their passwords in easy to find locations. For the system administrators, additional applications result in an increased directory maintenance effort and fielding increased help desk calls to reset passwords. Some of the problems that are caused by multiple applications can be resolved by using single sign-on (SSO).

Note: Secure refers to applications that require users to be authenticated before they can access the application.

The implementation of an SSO solution is the responsibility of the custom implementation. It is recommended that an IBM or third-party tool is used. For example, IBM Tivoli tools or CA SiteMinder.

Single Sign On with WebSphere

When SSO is required with WebSphere, it can be achieved using the WebSphere lightweight third-party authentication mechanism (LTPA) and additional custom login modules. The LTPA protocol results in a token being created for an authenticated user. In WebSphere, a token is generated once credentials are added for an authenticated user. This token is then used to retrieve identity information for an authenticated user in an SSO environment.

Security is implemented as a Cúram login module within a chain of login modules set up in WebSphere. It is expected that at least one of these login modules be responsible for adding credentials for the user. By default, the Cúram login module adds credentials for an authenticated user. As a result of this, the configured WebSphere user registry handled by a subsequent login module does not add credentials. The recommended approach to implementing an SSO solution is to add a custom login module somewhere along the chain of login modules.

The ability to disable the addition of credentials for an unauthenticated user is provided, thus enabling an SSO solution to be implemented.

The Cúram JAAS login module for WebSphere checks if an LTPA token exists within WebSphere using the WSCredTokenCallbackImpl callback for WebSphere. If this token exists and is valid, then no authentication is performed by the Cúram login module.

Credentials may be added to the WebSphere user registry. Credentials include authentication information on the user logging in, including the unique identifier for the user. WebSphere checks that credentials exist for a user after all configured system login modules have executed, if the credentials exist, then the WebSphere user registry is not queried. Credentials are not added by the Cúram JAAS login module if the following settings are in place:

- curam.security.check.identity.only property is set to true.
- curam.security.user.registry.enabled property is set to true.

As mentioned in "Deployment of an External Application" on page 25, there are properties relating to the type of external user that control if credentials are added to WebSphere for a specific external user type. These include:

- curam.security.user.registry.enabled.types property.
- curam.security.user.registry.disabled.types property.

These properties provide fine grained control over authentication for external user types.

In the case where the Cúram JAAS login module does not add credentials, the WebSphere user registry will be queried to attempt to add credentials for the user.

Single Sign On for WebLogic Server

When SSO is required with WebLogic Server , it can be achieved by using the WebLogic Server authentication provider or a custom authentication provider. Consult the WebLogic Server documentation for further information on authentication providers. WebLogic Server expects credentials/principals and the group the user belongs to, to be added by the configured authentication provider. For an SSO solution the Cúram JAAS login module does not add credentials to the JAAS subject to allow for an alternative authentication provider to be responsible for adding credentials.

Credentials are not added if the following settings are in place:

- curam.security.check.identity.only is set to true.
- curam.security.user.registry.enabled is set to true.

As mentioned in "Deployment of an External Application" on page 25, there are properties relating to the type of external user that control if credentials are added to WebLogic Server for a specific external user type. These include:

- curam.security.user.registry.enabled.types property.
- curam.security.user.registry.disabled.types property.

These properties provide fine grained control over authentication for external user types.

The responsibility for adding credentials is left to another authentication provider, i.e., the main authentication provider for authenticating the user. In an SSO scenario, only one of the authentication providers needs to add credentials to the JAAS subject during the commit() method of the login module for a user

Other Security Considerations

Another important security concern is protecting content as it is entered, displayed, and transferred across the network for the Cúram application. The default configuration uses SSL provided by the application server to secure content as it is transferred.

In addition to this protection, industry-leading products are used during the development lifecycle to regularly monitor for security vulnerabilities in the application. Examples of such potential vulnerabilities include cross-site scripting, and SQL injection. Such threats are resolved within the infrastructure when discovered.

For the best security, customers must do similar security monitoring of their application.

SSL Settings for the Application

SSL is on by default for access to the web application. This ensures a secure SSL connection between the client and server and also ensures data is encrypted. SSL is turned on for the client through settings in the `web.xml` file for the web client application.

SSL is turned on at the application server level by settings in WebLogic Server and WebSphere . These settings for the application servers are done through the Cúram configuration scripts.

Important: The configuration scripts ensure SSL is turned on by default, however, this is a default configuration that must be updated and new certificates must be established for the SSL protocol.

It is recommended to leave SSL on for access to the Cúram application, however depending on specific project configurations, there may be a need to turn SSL off for the application.

It is possible, but not recommended to turn off SSL. "Turning Off SSL Settings for the Application" on page 31 should be consulted for further details.

Using Cúram in a Secure Environment

Cúram can be used in a secure server environment (e.g. FIPS-compliant) and is dependent on the requirements and capabilities of that environment (e.g. WebSphere FIPS configuration). However there are a few specific areas where Cúram-specific or related operation or configuration is required:

- When using the DB-to-JMS feature, which is enabled via the `curam.batchlauncher.dbtojms.notification.ssl` property, described in the *Cúram Batch Processing Guide*
- When using the Word Integration Control, used for the FILE_EDIT widget, documented in the *Cúram Web Client Reference Manual*, which has two aspects to consider:
  - When needing to use it with a browser in a TLS v1.2 environment, which is discussed in the "User Machine Configuration" topic of the *Cúram Web Client Reference Manual*.
  - The SP800-131a-compliant version of the supporting jar file can be used as long as your browser JVM supports SHA2, regardless of whether the server environment supports SP800-131a. To digitally sign the Word Integration jar for SP800-131a compliance you must build your environment using the `enable-sha-2-signed-jars` property (e.g. `-Denable-sha-2-signed-jars=true`) when invoking the Cúram build targets (e.g. server, client, websphereEAR).

Client Security Considerations

Errors that occur on the client will result in HTML error pages being displayed. The HTML error pages, by default, will contain a Java exception stack trace of the errors that have occurred. This stack trace output is used in a development environment for debugging purposes. However, as the HTML error pages that contain the Java exception stack trace are not subject to the Cúram's application malicious code and filtering checks, they could potentially leave the application open to injection attacks, e.g. Cross-site scripting and link injection. To control this, the client property `errorpage.stacktrace.output` exists to determine if the Java stack trace should be written to the HTML error pages.

The property `errorpage.stacktrace.output`, when set to `true`, writes the Java exception stack trace to the HTML error pages. This property is set to `true` by default, however this property must be set to `false` in a production environment to avoid any security vulnerabilities. Please consult the *Cúram Web Client Reference Manual* for further details on this property.

Customizing Authentication

You can use the following customization points and development artifacts to customize Cúram authentication.

Customizing the Login Page

The default out-of-box login screen is represented by the `logon.jsp` file located in the `lib/curam/web/jsp` directory of the Client Development Environment for Java

(CDEJ). The `logon.jsp` file can be customized by creating a copy of the out-of-the-box file and placing this in a `webclient/components/<custom>/WebContent` folder, where <*custom*> represents the name of the custom web client component.

The section on Login Pages in the *Cúram Web Client Reference Manual* has guidelines on what needs to remain in place in the `logon.jsp` file and should be referenced for further details.

Applying Styling to the Login Page

Styling changes can be applied to the `logon.jsp` in the usual way, i.e., by adding the relevant CSS to any .css file in the custom component. The *Cúram Web Client Reference Manual* should be consulted for details on styling.

Enabling Usernames With Extended Characters for WebLogic Server

If the WebLogic Server application server is not being used, this section can be ignored.

If you have Cúram user names or passwords with extended characters (e.g. "üßer") WebLogic Server provides a proprietary attribute, **j_character_encoding** , which must be added to the **logon.jsp** form-based login page. The WebLogic Server documentation should be consulted for more information. The attribute must be added to the table element in the `logon.jsp` file, as shown.

```
<input type="hidden" name="j_character_encoding" value="UTF-8"/>
```

Changing the Case-Sensitivity of the Username

The curam.security.casesensitive property controls the case sensitivity of usernames. By default, this is set to `true` in the `Application.prx` file. When set to `false` in the `Application.prx` file, this will result in the authentication and authorization mechanisms ignoring the case of the username.

The *Cúram Configuration Settings* chapter in the *Cúram Server Developer's Guide* should be consulted for further details on the `Application.prx` file.

Adding Custom Verifications to the Authentication Process

To add custom verifications, the `curam.util.security.CustomAuthenticator` interface must be implemented. This interface contains one method - `authenticateUser()` . The `authenticateUser()` method is invoked for both default authentication and identity only authentication. The results of this method are expected to be an entry from the `curam.util.codetable.SECURITYSTATUS` codetable. In the case of successful authentication, the result must be `curam.util.codetable.SECURITYSTATUS.LOGIN`

For authentication failures anything, including null, can be returned. It is recommended though that another code from the `curam.util.codetable.SECURITYSTATUS` codetable be used. This codetable can be extended to include custom codes as detailed in the chapter on Code Tables in the *Cúram Server Developer's Guide*.

After the custom verifications are invoked, the authentication process will update the relevant fields on the Users database table. For example, if the result of the customized verifications is not `SECURITYSTATUS.LOGIN` the number of login failures is increased by 1, and if the break-in threshold is reached, the account will be disabled. Alternatively, if the result is `SECURITYSTATUS.LOGIN` , the login failures are reset to 0 and the last successful login field is updated.

Note: When identity-only authentication is enabled the fields of the Users database table are not updated, irrespective of the result of the custom verification.

Configuring the Custom Authenticator

To configure the application to use this custom extension, the property curam.custom.authentication.implementation in the `Application.prx` must be set to the fully qualified name of the class implementing the `CustomAuthenticator` interface.

The *Cúram Configuration Settings* chapter in the *Cúram Server Developer's Guide* should be consulted for further details on the `Application.prx` file.

Configuring Identity Only Authentication

To configure identity-only authentication the curam.security.check.identity.only property should be set to `true` in the `AppServer.properties` file before running the **`configure`** target. It is also possible to set this property once the application is deployed through the application server console. For more information on configuring the application server the *Cúram Server Deployment Guides* for the application server being used should be consulted.

Adding the Cache Refresh Failure Callback Interface

The new callback class must implement the interface: `curam.util.security.SecurityCacheFailureCallback` in a class that has a public default constructor. The implementation of the callback is registered by setting the application property curam.security.cache.failure.callback to the name of the implementation class. If the property is not set, no attempt is made to invoke a callback handler.

Turning Off SSL Settings for the Application

SSL is on by default for access to the Cúram application. This ensures a secure SSL connection between the client and server and also ensures data is encrypted. SSL can be turned on and off for the client through settings in the `web.xml` file for the web client application, and at the application server level by settings in WebLogic Server and WebSphere . These settings for the application servers are configured via the configuration scripts. It is recommended to leave SSL on for access to the application, however depending on specific project configurations, there may be a need to turn SSL off for the application. The following sections detail how to do this.

Modifying the web.xml File for the Client Application

This can be modified by changing the <transport-guarantee> from `CONFIDENTIAL` to `NONE` in the `web.xml` file. Note, this does not disable access to the web client over HTTPS, but enables additional access via HTTP. For further details on modifying the `web.xml` file, the section on *Customizing the Web Application Descriptor* in the *Cúram Web Client Reference Manual* should be referenced. An example of setting this property is shown.

```
<user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
```

Modifying the Application Server Configuration

Modifying the configuration for WebSphere can be done in one of two ways. The first approach below being the recommended approach.

- Use the existing non-secure port, setup by default for Web Services (recommended approach). This caters for both SSL and non-SSL connections.
  1. Navigate to Environment -> Virtual Hosts -> client_host->Host aliases
  2. Click New and enter * for host name and 9082 for port number, then click OK
  3. On the next page click Save to store your new value to the server configuration. Please note that the port 9082 corresponds to the *CuramWebServicesChain* configured in the default client application and this port is now the port that can be used to access the application using HTTP
- Reuse the current SSL port of 9044 :

  The current port can be set up as a non-secure port. The steps to do this are described in the *Cúram Deployment Guide for WebSphere Application Server* - Section A.2.11 Server Configuration - Set up port access. Follow Steps 7 to 11 inclusive. The only difference for Step 11, is that the Transport Chain Template should be set to 'WebContainer' (and not WebContainer Secure).
- Complete the below steps after following any of the above step, to turn of SSL in Global Security Settings :
  1. Navigate to Security -> GlobalSecurity ->
  2. Select Web and SIP Security -> Single Sign-On (SSO)
  3. UnTick requires SSL , then click OK, save the server configuration.

Analyzing the AuthenticationLog Database Table

All authentication attempts (both successes and failures) are logged in the AuthenticationLog database table. The following are the rows of interest on this table:

Table 2. Contents of the Authentication Log

Field	Meaning
timeEntered	The timestamp of the entry in the log.
userName	The username associated with the login attempt.
altLogin	Boolean indication of whether the username represents an alternate Login ID. When this column equals '1' (true) the value in the userName column is an alternate login ID as per “Alternate Login IDs” on page 3; otherwise, the userName column represents the userName from the Users or ExternalUser table.
loginFailures	The number of login failures for this user since their last successful login.
lastLogin	The date and time of the last successful login.

Table 2. Contents of the Authentication Log (continued)

Field	**Meaning**
loginStatus	The status of the login attempt. This may be one of: • LOGIN: Successful login. • ACCDISABLE: The account has been explicitly disabled. • ACCEXPIRED: The password expiry date has been reached. • PWDEXPIRED: The number of days which the user was given to change their password has been exceeded. • BADUSER: The user does not exist. • AUTHONLY: This is used in the case of identity only authentication and indicates that only authorization verifications will be performed. • BADPWD: The specified password was incorrect. • BREAKIN: A specified number of incorrect passwords has been reached. The account is disabled. • RESTRICTED: The user is not allowed access the system at this time. • LOGEXPR: The number of login attempts which the user was given to change their password has been exceeded. • AMBIGUOUS: The specified username is ambiguous as it is a case insensitive duplicate of another username.

The `LogAdmin` API can be used to query the AuthenticationLog database table. The Java documentation for this class should be referenced for further details.

Customizing Authorization

Use this information to set up authorization for Cúram users.

Creating Authorization Data Mapping

The authorization data for a user can be set up through the use of the Data Manager (DMX files) or through the Cúram Administration screens. The *Cúram System Configuration Guide* should be consulted for details on identifying how to group security from a business perspective.

To create a new security role for a user, the security identifiers (SIDs) that the user must have access to, need to be identified. These SIDs should then be organized into groups of SIDs. The role, groups and SIDs, once identified, need to be set up on the security tables that these represent.

Security data is considered essential for the set up of a Cúram application. As such, the examples below describe adding security data to the `data/initial` directory within the component.

Creating a New Security Role

To create a new security role, a new entry must be added to the SecurityRole database table, setting the `rolename` attribute.

To do this, create/add to the `SecurityRole.dmx` file in the `%SERVER_DIR%/components/<custom>/data/initial` , where `<custom>` is any new directory created under components that conforms to the same directory structure as `components/core`.

Creating a New Security Group

To create a new security group, a new entry must be added to the SecurityGroup database table setting the `groupname` attribute.

To do this, create/add to the `SecurityGroup.dmx` file in the `%SERVER_DIR%/components/<custom>/data/initial` , where `<custom>` is any new directory created under components that conforms to the same directory structure as `components/core`.

Linking the Security Group to the Security Role

The security role must be linked to the security group. To do this, create a new entry in the SecurityRoleGroup table, setting the `rolename` and `groupname` attributes.

To do this, create/add to the `SecurityRoleGroup.dmx` file in the `%SERVER_DIR%/components/<custom>/data/initial` , where `<custom>` is any new directory created under components that conforms to the same directory structure as `components/core`.

Creating the Security Identifier (SID)

The create a new SID, an entry must be added to the SecurityIdentifier table, setting the `sidname` and `sidtype` attributes.

To do this, create/add to the `SecurityIdentifier.dmx` file in the `%SERVER_DIR%/components/<custom>/data/initial` , where `<custom>` is any new directory created under components that conforms to the same directory structure as `components/core`.

Linking the Security Group to the SID

To link the security group with the SID, an entry must be added to the SecurityGroupSID table, setting the `groupname` and `sidname` attributes.

To do this, create/add to the `SecurityGroupSID.dmx` file in the `%SERVER_DIR%/components/<custom>/data/initial` , where `<custom>` is any new directory created under components that conforms to the same directory structure as `components/core`.

Linking the Security Role to the User

To associate authorization data to a user, the security role must be linked to the user.

To do this, update the entry for the specified user in the `Users.dmx` file located in the `%SERVER_DIR%/components/<custom>/data/initial` , where `<custom>` is any new directory created under components that conforms to the same directory structure as `components/core` , setting the `rolename` attribute to be the `rolename` as specified on the SecurityRole table.

Loading Security Information onto the Database

Once all of the information has been entered in the various DMX files, the Data Manager should be used to load the DMX data onto the database. The *Data Manager* chapter in the *Cúram Server Developer's Guide* should be consulted for further details.

Creating Function Identifiers (FIDs)

When a method is made publicly accessible; by setting the stereotype to be <<facade>>, security is automatically switched on. This means a SID is automatically generated for that method and the security enabled flag for the method is set to `true` . The SID and its `fidenabled` flag are stored in the database-independent `<ProjectName>_Fids.xml` file located in the `/build/svr/gen/ddl` subdirectory. This file is used to insert the FID information onto the database via the Data Manager.

A FID follows the naming convention of `<classname>.<methodname>` , and the maximum length of a FID is 100 characters. For example, for a BPO called `ProductEligibility` , with two methods called `insertProduct` and `testProduct` , two FIDs are created: `ProductEligibility.insertProduct` and `ProductEligibility.testProduct`.

If security for a process method is switched off at design time in the model, a SID/FID is still generated but the security enabled flag is set to `false` . Setting the security enabled flag to `false` means that no authorization check is performed for this method.

Switching Security off for a Process Method

Setting the `Generate_Security` option on the process method to `false` in the model switches off security for a process method.

If security for a process method is switched off at design time in the model, a FID is still generated but the security enabled flag is set to `false` . Setting the security enabled flag to false means that no authorization check is performed for this method.

Security Considerations During Development

It is important to consider the effect of these design options when implementing security during the development of a Cúram application. They are the first and last line of defense against unauthorized access to application process functionality. Generally speaking, security will be switched on for almost all process methods. Security may be switched off for a process method that does not need security, e.g., a login method that gets invoked when a user tries to login to an application. As a user has not yet been authenticated or authorized, they need access to this method in order to login, therefore switching off security for this method may be necessary.

During the initial design phase of an application the overhead of keeping the security environment "in sync" with an evolving application can be tedious. It is possible to disable the authorization check by setting the curam.security.disable.authorisation property in the Application.prx file.

warning: Warning

The curam.security.disable.authorisation property should only be turned on at design phase. This should never be set to `true` in a production environment.

Finally, it should be noted that once the code and scripts have been generated from a working model, the information associated with a FID cannot be changed. To change this information requires modifying the model, re-generating and re-building the database.

Controlling the Logging of Authorization Failures for the Client

By default, web client authorization failures are not recorded.

The curam.enable.logging.client.authcheck property controls whether the authorization failures encountered by the web client are logged or not. This property is `false` by default, meaning these failures will not be logged. When set to `true` a log of these authorization failures is stored on the database table AuthorisationLog . The *Cúram Server Developers Guide* , Application.prx - Dynamic properties section should be consulted for more information on this property.

Authorizing New SID Types

A server interface method is provided to enable authorization to be performed directly. This method may be added to a class that manipulates data on the conceptual element being secured by the new SID type.

`curam.util.security.Authorisation.isSIDAuthorised()`

A usage example of the isSIDAuthorised() method is below:

```
// The SID associated with the conceptual element
       // to be secured.
       String someSID = "someSID";

       // Get the logged in username
       String loggedUser =
         curam.util.transaction.TransactionInfo.getProgramUser();

       // Check if the user has access rights
       if (curam.util.security.Authorisation.isSIDAuthorised(
             someSID, loggedUser)) {
         // Do something sensitive that this user has rights to do
         ...
       } else {
       // Throw an exception indicating the user doesn't have
       // access to perform this action
         AppException exception
           = new AppException(MESSAGE.ERR_USER_NO_ACCESS);
         throw exception;
       }
```

Analyzing the AuthorisationLog Database Table

All authorization failures are logged in a database table called the AuthorisationLog. The following are the rows of interest on this table:

Table 3. Contents of the Authorization Log

Field	**Meaning**
timeEntered	The timestamp of the entry in the log.
userName	The username associated with the authorization attempt.
identifierName	The security identifier (SID) or functional identifier (FID) associated with the failure.

The `LogAdmin` API can be used to query the AuthorisationLog database table. The Java documentation for this class should be referenced for further details.

Customizing Cryptography

Use this information to configure and customize cryptography for Cúram.

Cipher Customization

Modification of the default cipher settings is a relatively straightforward process, but needs to be adequately planned and tested. You will require an application restart for the changes to be implemented and depending on the size and topology of your organization and deployments you need to choose a time when in-progress changes won't be an impact. Also, consider any data (e.g., properties containing encrypted passwords) managed by the Cúram Transport Manager (CTM) that will either need to be updated or managed to prevent systems from being out of sync with one another (see the *Cúram Transport Manager Guide* for more information).

Modification of the default cipher settings involves the following steps:

1. Choosing new settings for the `CryptoConfig.properties` and underlying artifacts - see "Cúram Cipher Settings" on page 18
2. Depending on the settings, you may need to perform additional steps (e.g. when modifying the keystore as per "How to Create a New Keystore" on page 38).
3. Modify the `CryptoConfig.properties` file; note the default location is `<SERVER_DIR>/project/properties`.
4. Remove any existing `CryptoConfig.jar` files (these contain `CryptoConfig.properties`) that are found in the `<JAVA_HOME>/jre/lib/ext` directory (`$JAVA_HOME/lib/ext` on IBM® z/OS®). If any Cúram clients or servers are running these will need to be terminated in order to be able to deploy an updated `CryptoConfig.jar` file with the updated settings.
5. Re-encrypt the passwords in all existing property files as identified in "Cipher-Encrypted Passwords" on page 20. The Apache Ant configtest, configure, and installapp targets will place an updated `CryptoConfig.jar` file in the Java `lib/ext` directory.
6. Test and verify your changes.

Testing of your changes should include verifying any functionality that would be impacted; for example:

- Ensure the Ant configtest target still works.
- Ensure batch programs still work.
- If you utilize the Ant configure target ensure it still works.

Related topics:

- "Cúram Digest Settings" on page 19
- "Cipher-Encrypted Passwords" on page 20

Key Management

The management of the secret key for Cúram encrypted passwords is done via the JDK-provided **`keytool`** command, or equivalent. You will need to make local decisions about placement and isolation of the secret key for Cúram that are compatible with your local organization and standards.

Keep in mind that some settings passed to the **`keytool`** command need to be reflected in the `CryptoConfig.properties` settings, which needs to be coordinated for successful deployment as discussed in "Cipher Customization" on page 37. The following table shows the relationship between **`keytool`** command arguments and the Cúram crypto properties.

Table 4. Relationship of keytool Command Arguments to Cúram Crypto Properties

Keytool argument	CryptoConfig.properties property
-keyalg	curam.security.crypto.cipher.algorithm
-alias	curam.security.crypto.cipher.keystore.seckey.alias
-keystore	curam.security.crypto.cipher.keystore.location
-storepass	curam.security.crypto.cipher.keystore.storepass

Note: The secret key password defaults to the storepass password and should not be changed.

See the JDK documentation for more information on using the **`keytool`** command.

Related topics:
- "Cúram Cipher Settings" on page 18
- "Cryptography Properties" on page 18
- "How to Create a New Keystore"

How to Create a New Keystore

Creating a new keystore to replace the Cúram default requires running the **`keytool`** command provided with the JDK (or equivalent), modifying the CryptoConfig.properties settings to correspond (necessary, only if the keystore name and/or location is changed from the default, but changing the name can make your customizations more obvious), and ensure the Curam Ant targets can find the new keystore (necessary, only if the default location is changed).

For example:

```
keytool -genseckey -v  -alias MySecretKey -keyalg AES -keysize 128
-keystore MyOrganization.keystore -storepass secretpw -storetype jceks
```

The section "Key Management" on page 37 identifies the **`keytool`** command arguments that relate to the `CryptoConfig.properties` settings.

The default location of the keystore file is the `<SERVER_DIR>/project/properties` directory with a sub-directory structure that reflects the JDK in use: "ibm" for the IBM JDK and "sun" for the Oracle JDK. So, when creating a keystore file the Curam build scripts expect to find it in the case of the IBM JDK in: `<SERVER_DIR>/project/properties/ibm`. If you desire to use a location different from the default you can do one of two things:

1. Use an absolute location for the keystore file as described in "Cryptography Properties" on page 18. In this case the Curam default keystore files in `CryptoConfig.jar` will be ignored in favor of the absolute setting `CryptoConfig.properties`.
2. Use the Ant `crypto.prop.file.location` property when you run any of the targets, described in "Cipher Customization" on page 37, that create and copy the `CryptoConfig.jar` to point to your alternate location. The location specified will have to reflect the structure of your JDK - "ibm" or "sun". For instance:

- Place the new keystore file in a location like this on Windows for the IBM JDK: `C:\Curam\keystore\ibm\MyOrganization.keystore`
- Point to that location when running the build targets: `ant configure -Dcrypto.prop.file.location=C:\Curam\keystore`

Note: In the example above the change of keystore file name to `MyOrganization.keystore` will require a corresponding change to `CryptoConfig.properties` as per "Cryptography Properties" on page 18.

Note: The only supported keystore type for Cúram cryptography is jceks.

Following the keystore creation you need to follow the steps in "Cipher Customization" on page 37.

Related topics:
- "Key Management" on page 37
- "Cipher Customization" on page 37

Digest Customization

Modification of the default digest settings is a relatively straightforward process, but needs to be adequately planned and tested. You will require an application restart for the changes to be implemented and depending on the size and topology of your organization and deployments you need to choose a time when in-progress changes won't be an impact. Also, consider any data (e.g., User passwords) managed by the Cúram Transport Manager (CTM) that will either need to be updated or managed to prevent systems from being out of sync with one another (see the *Cúram Transport Manager Guide* for more information).

The process is covered in detail in "How to Utilize the Superseded Digest Settings for a Period of Migration" on page 40.

Related topics:
- "Cúram Digest Settings" on page 19
- "How to Specify a Digest Salt"

How to Specify a Digest Salt

While Cúram doesn't specify one out-of-the-box, specifying a salt for digested passwords provides an additional level of protection against brute-force attacks.

To specify a salt for your digested passwords:
1. Choose a sufficiently long and random string.
2. Encrypt this string using the Ant **`encrypt`** target (as documented in the *Cúram Server Developer's Guide*).
3. Place the encrypted string in a file.
4. Specify the location of the file containing the encrypted salt string using the `curam.security.crypto.digest.salt.location` property in `CryptoConfig.properties` and ensure that any deployed `CryptoConfig.jar` files reflect the updated settings.

For manageability you should make these changes in conjunction with the steps in "How to Utilize the Superseded Digest Settings for a Period of Migration" on page 40.

How to Utilize the Superseded Digest Settings for a Period of Migration

Utilizing the superseded digest settings means you are migrating your existing digested passwords to a new crypto configuration (e.g. new salt) and would like Cúram user passwords automatically migrated for a period of time. This applies to Cúram internal and external users, but does not apply to users managed by third-party security systems such as LDAP.

The process to do this is:

1. Choose a time when your Cúram system can be down and with the Cúram system not running.
2. Copy the existing digest property names and values in `CryptoConfig.properties` and rename the properties to the new superseded property names.
3. Modify the existing digest property names in `CryptoConfig.properties`.
4. Set the `curam.security.convertsupersededpassworddigests.enabled` property to '`true`'.
5. Set the `curam.security.crypto.upgrade.start` property to help you track when you introduced the updated configuration. This value can be used below to help manage unmigrated user passwords.
6. Restart the application server, but note the following.

Note: The Cúram default web services user (WEBSVCS), or any user not processed via the `CuramLoginModule`, is not available for automatic password migration. You must reset these users before restarting the application server. To do this:

1. Obtain the new digest password value via the Ant digest target (e.g. `ant digest -Dpassword=password`).
2. Update the password value in the database, which is easily done via SQL (e.g. `UPDATE USERS SET PASSWORD='<new digest value>' WHERE USERNAME='WEBSVCS';`).
3. You can now start the application server

After a period of time (e.g. weeks or months) when you consider the migration period to be over set the `curam.security.convertsupersededpassworddigests.enabled` property to '`false`' and unset the `curam.security.crypto.upgrade.start property`.

Users who did not login during the migration period will now see their logins fail due to password mismatches. You have two approaches for addressing the passwords not updated during the migration period:

1. Require these users to contact your internal support to have their password reset via the admin user interface.
2. Manually identify the users in the Cúram USERS table who were not updated during the migration period and either manually set new default password either via SQL (see the **`digest`** target described in the *Cúram Server Developer's Guide* to obtain new digest password values) or via the admin user screens. For example, using the following query: `SELECT username FROM users WHERE lastwritten between timestamp('2013-06-01 15:00:00') AND timestamp('2013-09-01 00:00:00')`

You should not leave `curam.security.convertsupersededpassworddigests.enabled` set to true indefinitely because:

1. It's meaningless to have gone to the trouble of upgrading from configuration 'A' to configuration 'B' and leave the original 'A' configuration active;
2. It leaves potentially weaker crypto settings active in the system; and
3. In order to use this functionality for a future upgrade, say from configuration 'B' to 'C', you would have to have upgraded all the 'A' passwords to at least 'B'.

Note: Any files, e.g. DMX, with stored digests need to be considered with respect to your migration strategy so they reflect the correct values.

Note: Any use of the Cúram Transport Manager (CTM) during a migration needs to be considered in terms of ensuring compatible settings and expectations between the source and target systems.

Related topics:
- "Cúram Cipher Settings" on page 18
- "Cúram Digest Settings" on page 19

Modifying Your Crypto Configuration for a Production System

While the out-of-the-box (OOTB) crypto settings are adequate for typical development or test environments, they should be modified for production environments to protect and provide isolation between these relatively low-risk environments and high-risk production environments.

Some typical changes to the OOTB crypto configuration, in preparation for production, might include:
- Providing a new secret key.
  - Such a key can be generated using the JDK keytool utility; see "How to Create a New Keystore" on page 38
    - This secret key should be stored in a separate keystore.
    - The properties for these secret key changes would be as described in "Key Management" on page 37.
- Providing new digest settings
  - New digest settings can include a new salt, iteration count, and/or algorithm.
    - The properties for these digest changes would be as described in "Cúram Digest Settings" on page 19 and "How to Specify a Digest Salt" on page 39 and the process described in "How to Utilize the Superseded Digest Settings for a Period of Migration" on page 40.

Remember to keep your configuration files isolated from personnel who do not absolutely have to access; specifically, keeping development, test, and production configuration information isolated.

Customizing External User Applications

Use this information to customize external user applications. As external users are processed differently to internal users, a separate Cúram web application is required specifically for external users.

Creating an External User Application

A new web client application must be developed for external users. The *Cúram Web Client Reference Manual* should be consulted for details on creating a new web client application.

Creating an External User Client Login Page

A new logon.jsp must be created for an external user application. The Cúram Platform ships with a default login page, `logon.jsp` , located in the `lib/curam/web/jsp` directory of the CDEJ (Client Development Environment for Java). This file should be copied to a `webclient/components/<custom component>/WebContent` folder in the web client application and modified as follows:

The `table` element should be extended to include a hidden input field user_type:

```
<input type="hidden" name="user_type"
         value="EXTERNAL"/>
```

Where `EXTERNAL` indicates the type of external user. This can be set to any value, excluding `INTERNAL`.

Creating an External User Client Automatic Login Page

Some external user client applications require no user authentication and hence a username and password should not be requested. It is not possible to disable authentication in Cúram , so the best way to achieve this requirement is to write an automatic login script.

The automatic login script takes a hard coded username and password and provides that as the authentication information when requested. This means that all users for such an application will always execute under the same username. Use of such a script should be limited to true open access applications.

When implementing applications that have a need for an automatic login, the implications for session management must be considered. Session management in Cúram maintains a user's session information to ensure when the user logs back in, the relevant session information, i.e., their tabs and navigation opens to where they left off for them. In the case of a user that has been automatically logged in, this information must not be maintained, therefore session management may need to be turned off in this scenario. The *Cúram Web Client Reference Manual* should be referenced for further details on how to turn this off.

The following are examples of automatic login and logout JSP scripts.

Note: Security implementations and configurations differ across application server vendors so these examples may not work in all cases or for all application server versions.

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:prefix="URI"
  version="2.0">
  <jsp:directive.page buffer="32kb"
                      contentType="text/html; charset=UTF-8"
                      pageEncoding="UTF-8" />
  <jsp:text>
    <![CDATA[
      <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">]]>
  </jsp:text>

  <!-- Automatic redirect to login security check of user
         details specified below -->

  <html>
    <head>
```

```
        <script type="text/javascript">
          function autoSubmit() {
            document.getElementById("loginform").submit();
          }
        </script>
        <meta content="text/html; charset=UTF-8"
              http-equiv="Content-Type" />
      </head>
      <body class="logonBody"
            style="visibility: hidden;"
            onload="autoSubmit()">
        <form id="loginform"
              name="loginform"
              action="j_security_check"
              method="post">
          <input type="hidden"
                 name="j_username"
                 value="generalpublic" />
          <input type="hidden"
                 name="j_password"
                 value="password" />
          <input type="hidden"
                 name="user_type"
                 value="EXTERNAL" />
        </form>
      </body>
    </html>
  </jsp:root>
```

Automatic Logout JSP

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:prefix="URI"
  version="2.0">
  <jsp:directive.page buffer="32kb"
                      contentType="text/html; charset=UTF-8"
                      pageEncoding="UTF-8" />
  <jsp:text>
    <![CDATA[
      <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">]]>
  </jsp:text>
  <html>
    <head>
      <script type="text/javascript">
        function autoSubmit() {
          document.getElementById("logout").submit();
        }
      </script>
      <meta content="text/html; charset=UTF-8"
            http-equiv="Content-Type" />
    </head>
    <body class="logoutBody"
          style="visibility: hidden;"
          onload="autoSubmit()">
      <form id="logout"
            name="logout"
            action="servlet/ApplicationController"
            method="post">
        <input type="submit"
               name="j_logout"
               value="Log Out" />
        <input type="hidden"
               name="logoutExitPage"
               value="redirect.jsp" />
      </form>
    </body>
  </html>
</jsp:root>
```

Extending the Public Access User Class

To “hook” the custom solution into the application the `curam.util.security.PublicAccessUser` abstract class must be extended, which requires implementing the `curam.util.security.ExternalAccessSecurity` interface. That concrete class will be used during the authentication and authorization process to determine required information relating to the external user. This class and its methods are described in detail below.

Authenticating an External User

The `authenticate()` method is responsible for authenticating an external user. It is invoked during the authentication process if the user is identifier as an external user. In the case of external users this method is invoked in place of the configured authentication.

Note: If an alternative authentication mechanism, e.g. LDAP, is configured, the external users must be able to authenticate against this mechanism.

```
/**
   * The implementation of this method should validate the identifier and
   * password and return the result of the validation. If the information is
   * valid, the codetable code SecurityStatus.LOGIN should be returned.
   *
   * @param identifier The identifier of the external user.
   * @param password The password as array of characters.
   * @param userType The type of external user.
   *
   * @return The status of the authentication in the form of a codetable code.
   *
   * @throws AppException Generic Exception Signature.
   * @throws InformationalException Generic Exception Signature.
   */

  public abstract String authenticate(String identifier,
    char[] password, String userType)
    throws AppException, InformationalException;
```

The input parameters to the method include an identifier, the digested password as an array of characters, and the type of the external user to be authenticated.

The `userType` parameter is intended to allow for support of multiple types of external users that require different authentication mechanisms. The use of this parameter depends on the custom implementation.

The expected result of this method will be an entry from the `curam.util.codetable.SECURITYSTATUS` codetable. In the case of successful authentication the result must be:

```
curam.util.codetable.SECURITYSTATUS.LOGIN
```

For authentication failures this codetable contains a number of entries, including `BADUSER` , `BADPWD` and `PWDEXPIRED` . This codetable can be extended to include custom codes as detailed in the *Cúram Server Developer's Guide*.

The authentication result returned by this method is automatically logged in the `AuthenticationLog` database table. For more information on this table see the *Cúram Server Developers Guide*.

The abstract class `PublicAccessUser` also defines the following abstract methods that any concrete subclass must implement:

- Method `upgradeSafePasswordValidation()` is required to allow for password comparison and is defined as follows:

  ```
  public final boolean upgradeSafePasswordValidation(
  final String userName,
  final String storedPasswordHash,
  final String plaintextPassword)
  ```

- Method `setPassword()` is to allow the implementor to persist the password (e.g. a new password) in the case of crypto upgrades. So this method gets called when the `upgradeSafePasswordValidation()` method is called. Here is the method definition:

  ```
  public abstract void setPassword(String username, String hashedPassword)
  throws AppException, InformationalException;
  ```

See the associated Javadoc of the `PublicAccessUser` class for more details regarding the above methods.

Determine External User Details

Details for an external user are retrieved by calling the `getLoginDetails()` method of the `curam.util.security.ExternalAccessSecurity` interface. These details are returned directly after authentication to direct the external user to the correct application homepage.

```
/**
 * The implementation of this method should retrieve the
 * details of the user required to redirect them to the correct
 * application page. This information includes the name of the
 * application home page for the user, the default locale for
 * the user and a list of warnings/messages for the user.
 *
 * @param identifier The identifier of the external user.
 *
 * @return The user details, including the application
 *         home page.
 *
 * @throws AppException Generic Exception Signature.
 * @throws InformationalException Generic Exception Signature.
 */
UserLoginDetails getLoginDetails(String identifier)
  throws AppException, InformationalException;
```

An instance of the `curam.util.security.UserLoginDetails` class must be created and returned from this method. The following information should be returned using this class:

- `UserLoginDetails . setApplicationCode(String code)`

  The code corresponding to the application homepage for the external user.

  This must be a valid entry in the `APPLICATION_CODE` codetable.
- `UserLoginDetails . setDefaultLocale(String defaultLocale)`

  The default locale for the external user.

  This is the locale the application will be displayed in by default for the external user.
- `UserLoginDetails . setFirstName(String firstName)`

The first name of the external user.

This will make the user's first name available for display in the user-message for an application banner.

- `UserLoginDetails . setSurname(String surname)`

  The surname of the external user.

  This will make the user's surname available for display in the user-message for an application banner.
- `UserLoginDetails . addInformationals(InformationalManager informationalManager)`

  Any informationals that must be displayed to the external user.

  The `curam.util.exception.InformationalManager` class can be used to create a number of informational or warning messages that will be displayed when the external user logs in. For example, a warning to let the external user know that their password is due to expire.

Authorizing an External User

The `getSecurityRole()` method is used during authorization to determine the security role associated with the external user. The security roles used for external users are configured in the same way as the security roles for internal users.

```
/**
 * The implementation of this method should return the security
 * role associated with the external user for authorization
 * purposes. If the user does not exist null should be
 * returned.
 *
 * @param identifier The identifier of the external user.
 *
 * @return The security role for authorization.
 *
 * @throws AppException Generic Exception Signature.
 * @throws InformationalException Generic Exception Signature.
 */
String getSecurityRole(String identifier)
  throws AppException, InformationalException;
```

The SDEJ will invoke an implementation of this method during the authorization process if the user does not exist in the security cache. Only internal users can exist in the security cache. This means that the identifiers used to identify external users must be unique and not conflict with usernames setup for internal users, unless the custom `UserScope` interface as described in "User Scope" on page 25, is implemented. Otherwise, if any usernames conflict the access rights assigned to the internal user will also be used for the external user.

If a role cannot be determined for the external user, null must be returned so that the SDEJ can report the authorization error correctly.

Determining the User Type

The `getUserType()` method is used to determine if a user is an external user.

```
/**
 * Return the type of the user. This is to allow support for
 * different types of external user. If there is only one
 * type of external user, simply return "EXTERNAL".
 *
 * @param identifier The identifier of the external user.
 *
 * @return The type of the external user.
 *
 * @throws AppException Generic Exception Signature.
 * @throws InformationalException Generic Exception Signature.
 */
 String getUserType(final String identifier)
   throws AppException, InformationalException;
```

The `getProgramUserType()` in `curam.util.transaction.TransactionInfo` will invoke this method to return the type of user if the user is not recognized as an internal user. For internal users "INTERNAL" is always returned.

For external users, there may be multiple types of external users, so this method should return the specific type of external user.

Preventing the Deletion of a Security Role: Role Usage Count

The `getRoleUsageCount()` method is used to prevent the deletion of a security role that is currently referenced by an external user.

```
/**
 * Return the number of users using a particular role. This
 * method is used to ensure that a role cannot be deleted when
 * it is in use by an external user.
 *
 * @param role The security role name.
 *
 * @return The number of users currently using the
 *         specified role.
 *
 * @throws AppException Generic Exception Signature.
 * @throws InformationalException Generic Exception Signature.
 */
 int getRoleUsageCount(String role)
   throws AppException, InformationalException;
```

Security roles that are referenced by any user, internal or external, cannot be removed. This method should return a number of 1 or more if any external users reference the specified role.

Retrieving a Registered Username

The `getRegisteredUserName()` method is used retrieve the correct case username, which may be independent of the username typed during login.

```
/**
 * Gets the correct casing for this user independent of mixed
 * case which may have been typed in by the logged in user.
 *
 * @param identifier The identifier of the external user,
 * whose casing may not match that of the persisted identifier
 * for the user.
 *
 * @return The actual case for this user, before its case has
 * been modified by external factors.
 *
 * @throws AppException Generic Exception Signature.
 * @throws InformationalException Generic Exception Signature.
 */
 public String getRegisteredUserName(final String identifier)
   throws AppException, InformationalException;
```

The default implementation for this method should return the username that has been provided. It is only if the curam.security.casesensitive has been set to false that this method may need to change the case of the username returned.

Note: Where the curam.security.casesensitive property has been set to false and is required for external users, it is the responsibility of all methods in this interface to handle any case specific requirements.

Reading User Preferences

The `getUserPreferenceSetID()` method is used to retrieve the user preference set ID associated with an external user. If no user preferences exist for an external user, then the default preferences will be used for the external user. The *User Preferences* chapter in the *Cúram Server Developer's Guide* should be referenced for further details on user preferences.

```
/**
 * This method is used to retrieve a set of user preferences
 * associated with an external user. The userPrefSetID is a
 * foreign key to the UserPreferenceInfo table.
 * The UserPreferenceInfo table contains information on
 * the user preferences.
 *
 * @param identifier The identifier of the external user.
 *
 * @return The userPrefSetID for the external user.
 *
 * @throws AppException Generic Exception Signature.
 * @throws InformationalException Generic Exception Signature.
 */
 String getUserPreferenceSetID(final String identifier)
   throws AppException, InformationalException;
```

The default implementation for this method should return the user preference set ID for the user preferences associated with an external user.

Modifying User Preferences

The `modifyUserPreferenceSetID()` method is used to update the external user details with a new set of user preferences. Please see User Preferences for further details on user preferences.

```
/**
 * This method updates the external user details with new user
 * preferences.
 *
 * @param userPreferenceSetID The ID for the user preferences.
 * @param username The identifier of the external user.
 *
 * @throws AppException Generic Exception Signature.
 * @throws InformationalException Generic Exception Signature.
 */
 void modifyUserPreferenceSetID(
  final String userPreferenceSetID, final String username)
     throws AppException, InformationalException;
```

The default implementation for this method should update the user preference set id associated with an external user.

Configuring External Access Security

The `curam.custom.externalaccess.implementation` property must be set in the `Application.prx` to indicate the fully qualified name of the class which implements the above interface.

Note: The curam.custom.externalaccess.implementation property is not dynamic, and if changed the application must be restarted before the change will take effect.

Determining if a User is Internal or External using the UserScope Interface

To support alternative methods for determining if a user is internal or external the custom interface UserScope is available. For example, even though usernames must be unique across the set of internal and external users, this custom interface can be implemented to allow duplicate usernames across internal and external applications in a limited way.

To provide a custom implementation for determining the type of user, the curam.util.security.UserScope interface must be implemented. This interface has one method isUserExternal() that determines the type of user. This method should return true if the user is considered external or false indicating the user is internal.

For example, an installation might have application1 deployed with userA, a Cúram internal user, and application2 deployed with userA being external (e.g. defined to LDAP). The ability for application1 to use internal userA and application2 to use external userA would be controlled by different properties. That is, Bootstrap.properties in properties.jar in the application1 EAR would have a different custom property setting from application2 EAR and the implementation of curam.util.security.UserScope.isUserExternal() would interrogate this setting to decide if the user is internal or external.

To specify a custom implementation of the UserScope interface the curam.custom.userscope.implementation property must be set in Application.prx. This should be set to the fully qualified name of the class that implements the UserScope interface.

Note: The curam.custom.userscope.implementation property is not dynamic, and if changed the application must be restarted before the change will take effect.

The isUserExternal() method of the UserScope interface is detailed in "User Type Determination."

User Type Determination

The isUserExternal() method is invoked anywhere in the application where the type of user is to be determined. This includes when the user logs into the application and when they attempt authorization to access secured elements of Cúram .

```
/**
 * The implementation of this method should determine the type of
 * User that is logged into the application. There are 2 types of
 * users: INTERNAL and EXTERNAL. If the user is an EXTERNAL user,
 * then this method should return true. If false is returned,
 * then the user is considered INTERNAL.
 *
 * @param username - The username.
 * @return A boolean value of true indicating an EXTERNAL user,
 * false indicates an INTERNAL user.
 *
 * @throws AppException Generic Exception Signature.
 * @throws InformationalException Generic Exception Signature.
 */
boolean isUserExternal(String username)
  throws AppException, InformationalException;
```

Notices

Privacy Policy considerations

Programming Interface Information

This publication documents intended programming interfaces that allow the customer to write programs to obtain the services of IBM Cúram Social Program Management.

Trademarks

Printed in USA