IBM Cúram Social Program Management
Version 7.0.0

Cúram Universal Access Customization Guide

IBM

Revised: August 2014

This edition applies to IBM Cúram Social Program Management v6.0.5.5 interim fix 2 and to all subsequent releases unless otherwise indicated in new editions.

Contents

Figures

Tables

Customizing Universal Access

Use this information to customize the IBM Cúram Universal Access enterprise module. Universal Access consists of the CitizenWorkspace, CitizenWorkspaceAdmin, and WorkspaceServices components. The major customizable features are triage, screening, intake, security, the citizen account, and life events.

The Black Box Engineering Philosophy

Universal Access is a black box product. This means that it has been designed from the outset to be extremely flexible with the ability to change many aspects of its functionality at runtime simply through configuration. Many other aspects can be modified by using the UA APIs. If, after reading this information, you are still unable to do what you require, then you can request a feature in a Service Pack or other future release. By choosing this route the feature you are getting will be incorporated into the product with all the testing and quality assurance that this implies.

It Saves Time and Money

A black box engineered product can help save time and money. IBM is committed to responding to requests for enhancements in a satisfactory time frame. This ensures that you won't have to put time and expense into developing enhancements including all the support and expense that such work entails. IBM is also committed to developing the product where we see sensible enhancements and new configurations where they don't currently exist.

It Makes For Easier Upgrades

UA is a strategic platform for the deployment of social enterprise services to an agency's clients.

- Universal Access is a platform – it provides a set of APIs and extension points that can be used to build out a solution that suits the needs of individual customers.
- Universal Access is strategic – from the outset it has been built with upgrade concerns in mind. The goal is that upgrades are simple and each one brings in a host of new backward compatible features.

The second point is a key tenet of the Universal Access design philosophy: By using and extending Universal Access in the recommended ways, you can take on new versions with minimal effort and in doing so take advantage of all the new features offered through upgrades. If customers stray outside of the recommended guidelines set out in this document, then there is an increased risk of running into difficulties during an upgrade.

It Is Still Configurable and Customizable

Along with our commitment to the black box engineering philosophy, there are a number of customization and configuration options already in place. The UA Platform has been built to cover as many configuration points as possible out of the box, this information describes these options.

Configuring the application banner, menus & navigation

You can configure the application banner, including mega and other menus, and the navigation bar. The application banner is configured in the internal application configuration file, with the extension .app and you can reuse and extend this content to support the external applications. You can also use the existing navigation configuration files, with the extension .nav, to support the new external navigation bar.

Application Configuration

The full schema for the existing .app files can be found in the JDECommons/lib/schema folder, specifically the application-view.xsd file.

The following outlines the additional content that will be added or where existing content will be used in a different context.

Relevant attributes on the *application* element:

Table 1. Relevant attributes on the application element

Attribute	Description/Use	New/Existing
id	Existing attribute, which identifies the unique id of the application and must match the name of the file. For internal applications this is linked to the APPLICATION_CODE codetable and the users home page. This is not the case for external applications.	Existing
title	A reference to content in the .properties file. This content is not displayed on the application banner, but used by the administration screens to identify the internal application.	Existing
subtitle	A reference to content in the .properties file. This content is not displayed on the application banner, but used by the administration screens to identify the internal application.	Existing
mode	Where this is not set, it is assumed the .app file is for an internal style application. Where this is set to external, the only supported value for now, this indicates that the .app file defines an external style application. This will be used to handle content and validation differently for both styles of application .	New

All other attributes are unsupported and ignored for mode="external" .app files.

The following are new elements that will be supported as direct children of the *application* element. All elements are optional.

- *landing-page*

  The icon and text displayed on the left corner of the application banner, including the hyperlink to what is known as the landing page. When an application is first loaded, it is this page that is opened, if defined. This is unlike an internal application which uses the users APPLICATION_CODE value. Note: If no page-id or landing-page element is specified, then the first entry in the navigation will be used as the landing page. It is possible for no landing-page element to be defined and the renderer will display nothing in this case.

Table 2. Relevant attributes on the landing-page element

Attribute	Description	Required
title	A reference to the text displayed under the icon.	Yes
page-id	The page to open when the icon/text is clicked.	Yes

Table 2. Relevant attributes on the landing-page element (continued)

Attribute	Description	Required
icon	A reference to the image icon to display.	No

- *banner-menu*

  There are three types of banner-menu supported in the application banner:

  – mega - A mega main, which is the first menu displayed.

  – person - The person menu, which usually displays the users name/welcome message and options to logout.

  – help - A menu of help/contact links. This is similar in style to the mega-menu.

  It is possible to have no banner-menu elements and the renderer will display nothing in this case. In addition, a banner-menu can have no menu-items and again the renderer will handle this. In other words, these elements are optional.

Table 3. Relevant attributes on the banner-menu element

Attribute	Description	Required
type	mega, help and person are the currently supported values. And only one of each can be defined.	Yes
title	A reference to the title text to display.	Yes
page-id	A reference to the page to open. This is optional and only supported for the person menu in the first version.	No

A *banner-menu* can have 0 to n *menu-item* elements as children, with the following attributes:

Table 4. Relevant attributes on the menu-item type

Attribute	Description	Required
id	A unique id for the menu item. This must be unique within the file.	Yes
icon	A reference to the icon to display.	No
title	A reference to the title text.	Yes
text	A reference to a longer description text.	No
page-id	A reference to the page to open.	Yes

A *menu-item* can have 0..n child elements called *param*. The params are hard-coded values that will be passed as parameters to the link associated with the menu-item. They will have the following attributes:

Table 5. Relevant attributes on the param type

Attribute	Description	Required
name	The name of the parameter to pass with the link.	Yes
value	A reference to the value of the parameter. This can be localized and if not, will be passed as is.	Yes

A param is being used to set the "motivation" of the link. So in most cases the name will be motivation with some value. The value is localizable, but if it does not exist in the .properties file, the value specified in the xml will be sent.

- *navigation*

  A reference to the navigation file (.nav), which contains the list of items to display in the navigation bar for the application. Navigation bars are usually defined at a tab level, but in this case it is for the full application.

Table 6. Relevant attributes on the navigation type

Attribute	Description	Required
id	A reference to the id of the .nav file, which is the name of the file, e.g. <id>.nav	Yes
width	A reference to the value to be used as the pixel width of the navigation bar. This is to allow for localisation configuration on a per application basis.	No

Navigation Configuration

The full schema for the existing .app files can be found in the JDECommons/lib/schema folder, specifically the application-view.xsd file.

The following outlines the additional content that will be added or where existing content will be used in a different context.

Relevant attributes on the *application* element:

Table 7. Relevant attributes on the application element

Attribute	Description/Use	New/Existing
id	The identifier for the navigation file, which must match the name of the file.	Existing
loader-registry	A list of loaders that can be used to dynamically control the display of the content.	Existing
nodes	The list of navigation items	Existing

The *nodes* element supports two children:

- *navigation-group*

  This child element is not applicable for an external application navigation.
- *navigation-page*

  1..n *navigation-page* elements can be added to the navigation. Each represents a link in the navigation bar. The applicable attributes are:

Table 8. Relevant attributes on the navigation-page element

Attribute	Description/Use	New/Existing
id	The unique id of the entry, used to prevent conflicts during contribution.	Existing
title	A reference to the title of the navigation item.	Existing
description	A reference to an admin description for the item.	Existing
visible	A boolean indicating if the item is visible or invisible by default.	Existing
dynamic	A boolean indicating if the item's visibility can be controlled by a loader.	Existing
page-id	A reference to the page to be opened when the item is clicked.	Existing

Table 8. Relevant attributes on the navigation-page element (continued)

Attribute	Description/Use	New/Existing
icon	A reference to the icon to display for the entry.	New

A new child element, *highlight*, will be supported under the *navigation-page* element, only within the context of an external application. This will contain one attribute, but 0..n entries can be defined:

Table 9. Relevant attributes on the highlight element

Attribute	Description
page-id	The id of a UIM page, which when displayed in the content area will result in this navigation item being highlighted.

All page-ids above refer to UIM pages. The section "How to Add a New Page to Citizen Account" on page 24 describes guidance around adding UIM pages to Universal Access.

Securing Universal Access

Use this information to understand the Universal Access Security Model and how to customize Universal Access securely in line with this model.

Background information for UA Security

Universal Access is designed to give citizens access to their most sensitive personal data over the Internet. Security must be a primary concern when developing citizen account customisations. All projects built on Universal Access must have highly focused on delivering security. This requires the project team to think of security from the very beginning rather just testing it at the end. It is recommended that all projects take at least the following steps to ensure the security of their delivery:

- Ensure that the project team are familiar with the principles of secure application development, and common vulnerabilities such as the OWASP Top Ten
- Develop a Threat Model and apply it
- Employ security experts to test everything from requirements to the finished deployment
- Plan for how the application will be used in public spaces like libraries and kiosks

The Universal Access Security Model

Universal Access has a number of different account types, in order to support both anonymous and registered users using the application. As users progress through their use of UA, they transition through a number of these different account types. The user types can be summarized as:

- The Public Citizen Account
- Anonymous Accounts
- Registered Accounts
- Linked Accounts

The Public Citizen Account

When the user views the front page of Universal Access they are automatically logged in under the `publiccitizen` account. This account only has access to the home page and pages that allow for entry or reset of passwords.

Anonymous Accounts

When the user clicks on a link to perform Triage, Screening or Intake, they are automatically logged out as `publiccitizen` and logged back in under an anonymous account with a randomly generated user name. This username can be used for the duration of a session during which the user might perform Triage, Screening and/or Intake. There are good security reasons for associating each individual session with a different generated account. One of the core principles of UA is that users should not have access to the data of other users. If all Intake and Screenings were performed using a single user account, `publiccitizen`, for example, then there is potential for one user to end up seeing data that has been entered by another user.

Registered Accounts

Accounts of this type are standard accounts created by citizens. Citizens can create accounts when they first arrive at the application, or during processes like screening or intake. These accounts differ from Anonymous accounts in that they allow citizens to continue previously saved Screenings, re-start Intake Applications that were previously unfinished and review or withdraw previously submitted Intake Applications.

Linked Accounts

The final account type is Linked Accounts. Linked Accounts are accounts that have been linked with an underlying Concern Role ID for a `Person` entity in Cúram. These users have access to detailed information about their benefits and cases in the Cúram system, via citizen account. Users with a linked account can submit Life Events such as "I Lost my Job" or "I got married". They also have access to information about benefit payments. Because of the sensitivity of this information, customers must ensure that they have a robust process for creating linked user accounts.

Some typical scenarios for linking are presented below. These are examples only, the actual processes for linking will be unique to each customer. A client requests a Citizen Account. The client is asked to present themselves at their local Social Welfare office with drivers license and other personal identification. The case worker, uses custom developed Cúram functionality to enter details for the new linked account after verifying the identity of the client.

A client creates a user account for Universal Access and submits an Intake Application. They are contacted by their case worker who asks them if they want access to more services using the Universal Access system. The client agrees and presents themselves at the local office with identification such as a passport. The case worker is able to link the client to the account they used to submit the Intake Application.

In both of these cases the case worker does not have access to the client's password. Instead, the linking process triggers a batch job that generates a letter, sent to the client's home address. The letter contains the password and a separate letter then contains an electronic code card. All of this functionality is developed by the customer however it is supported by UA APIs that allow a UA username to be linked to a Concern Role ID.

Continuing the above scenario, the client receives a letter from the Social Enterprise containing their initial password (in the case of the first scenario) and instructing them that a code card will arrive shortly. The code card arrives by post the next day and the client is able to log in to their Citizen Account. The login screen contains a username and password as before, however there are also additional authentication factors - The client must enter their date of birth, social security number and a code from their electronic code card. This is called Multi-Factor Authentication.

Authorization Roles and Groups

The various account types described above are assigned different authorization roles. These roles limit the methods that can be invoked. No additional permissions should be granted to UA authorization roles except for Linked accounts, which use the LINKEDCITIZENROLE. If adding additional custom methods to citizen account, additional permissions will be required. For more information, see "Customizing the citizen account".

If only a subset of the functionality offered by UA is being used, permission to invoke the unused methods should be removed from the database. For example, if citizen account is not being used, the LINKEDCITIZENROLE and other related authorization artifacts should be removed, as they are not needed. Projects not using citizen account should also consider the deployment implications. For more information, see "Customizing the citizen account".

Authorization roles should be configured only for the areas of functionality that are actually being used. It is recommended that unused SIDs should be removed from the database. For example, if citizen account is not being used, the LINKEDCITIZENROLE and other related authorization artifacts should be removed, as they are not needed. Projects not using citizen account should also consider the deployment implications. For more information, see "Citizen Account Security Considerations".

Proper use of the UA Authorization Roles and Groups will ensure that no user can access functions for which they have no permission. It will not however, prevent users from using these functions to access data belonging to user users. This is the preserve of Data-based Security. UA provides a framework for Data-based Security and all customizations should use this framework. For more information, see "Citizen Account Security Considerations".

Deployment Considerations

Client components can be divided into those that form part of internal applications, and those that form part of public facing applications (such as UA). Components that contain artifacts intended for use in internal applications should not be deployed into public facing applications such as UA. Customisations should be split between internal and public facing client components in order to achieve this. Internal components should never be added to the UA deployment packaging, as this will mean that artifacts intended for case workers or administrators will be deployed into the public facing application.

The UA client-side artifacts are divided between the `citizenworkspace` and `citizenaccount` client components. This is done for good reason: the `citizenaccount` component includes UIM pages that expose sensitive data to citizens (including life events functionality), whereas the `citizenworkspace` component includes the artifacts needed to offer triage, screening and online application functionality. Accordingly, if the citizen account functionality is not being used, the `citizenaccount` client component should not be deployed, i.e. it should be removed from the UA deployment packaging. Please see the Cúram deployment guide related to your specific application server for more information on deployment and deployment packaging. For more information about securing the citizen account, see "Citizen Account Security Considerations".

Managing User names and Passwords

There is a range of ways to customize and configure the validations that are invoked when creating user accounts in UA. These can be used to enforce certain patterns on a username and password, for example, to prevent the username and password being identical, or to enforce a minimum number of characters for either.

Account Management

A description of the way you can customize account creation and management.

Account management configurations: A number of configurations properties are used to define the behavior of the validations in this area. Please see table below:

Table 10. Account configurations

Property	Description
`curam.citizenworkspace.username.min.length`	Minimum number of characters in the username.
`curam.citizenworkspace.password.min.length`	Minimum number of characters in the password.
`curam.citizenworkspace.password.min.special.chars`	Minimum number of special characters and/or numbers in the password.

The values of these configuration properties can be updated by logging in as sysadmin and selecting: Application Data->Property Administration. Select category "Citizen Portal - Configuration"

Account management events: Events are raised at key points during account processing. These can be used to add custom validations to the account management process. For more information on using events, please see the `Curam Server Developer Guide`. All of the following events can be found in the class `curam.citizenworkspace.security.impl.CitizenWorkspaceAccountEvents`

Table 11. Account events

Event Interface	Description
`CitizenWorkspaceCreateAccountEvents`	Events raised around account creation. For more, please see the related Javadoc information in the WorkspaceServices component.
`CitizenWorkspacePasswordChangedEvent`	Event raised when a user is changing their password. For more, please see the related Javadoc information in the WorkspaceServices component.
`CitizenWorksapceAccountAssociations`	Events raised when a user is linked or unlinked from an associated Person Participant. For more, please see the related Javadoc information in the WorkspaceServices component.

PasswordReuseStrategy API:

Customers are free to use the `curam.citizenworkspace.security.impl.PasswordReuseStrategy` API to add their own password change validations.

As part of the password reset function, Universal Access provides a default validation that prevents a user from entering a new password that is the same as the user's current password. Using the `PasswordReuseStrategy` API, custom validations can be added to restrict users from changing their passwords to current or previous values if required. For example, a customer might want to implement a password reuse strategy that prevents users from reusing a previous password until after six password changes.

For further details, see the API Javadoc.

CitizenWorkspaceAccountManager API: The `curam.citizenworkspace.security.impl.CitizenWorkspaceAccountManager` API is used to manage the creation and linking of UA accounts. It is envisaged that customers can use this API to build out custom functionality that supports case workers linking accounts, and creating accounts on behalf of the citizen. The API offers methods that support account management, including:

- Creating standard UA accounts
- Creating 'linked' UA accounts
- Removing links between Participants and accounts.
- Retrieving account information

Please see the API Javadoc for full details.

Data Caching

Customers need to be aware of the dangers posed by caching data in both browser and server caches. Care must be taken to minimize the risk of citizens being able to access each others' data from these caches. This can occur when the citizen uses the browser back button or history to retrieve data previously entered by other users, or when application PDF files are cached locally on the computer that was used to make the application.

HTTP Servers like Apache provide the ability to set cache-control response headers to not store a cache. We recommend this approach be taken with UA deployments to prevent access to data using the browser back button or history.

Browser Caching

Browsers can be configured never to cache content. If UA is to be offered in a "kiosk" or other publicly available guise, then the browser should be configured never to cache content.

Furthermore, it is advisable to customize UA in order to provide this guidance to citizens accessing the site via their own browsers. They should be advised to clear their cache and close all browser windows they have used when they are finished using UA. Citizens should also be made aware that PDF documents that they download from UA may need to be removed from the browser's temporary Internet files.

External Security Authentication

As an ever greater number of government services move to the Internet, there is a drive to ensure that citizens can be authenticated for any of these services using a single set of credentials. This provides benefits for the government in streamlining the authentication process and also for the citizen because they do not have to remember endless lists of usernames and passwords. This, in turn, increases security by making it less likely that citizens will write down their usernames and passwords and by focusing security efforts on implementing best practice in a single Enterprise Security System. In its Out-of-the-Box form, Universal Access uses its own authentication system which is backed up by a database of registered users. Universal Access can also be configured to integrate with External Security Systems.

Analysis

Consider this example analysis that is required in preparation for integration with an External Security System. Any analysis of requirements for External Security Integration should ask at least the following questions.

1. Is the Universal Access deployment to support anonymous Screening and/or Intake?
2. Is Account Management to be supported in Universal Access or in the External Security System? (for example, will account creation and password reset screens live in the External Security System or Universal Access).
3. Is Single Sign On Required?

Example UA customization requirements

In this example the team deploying Universal Access have the following requirements. This example will be used for reference when describing the configuration and development tasks.

1. Users can access Universal Access and perform anonymous Screening or Intake.
2. Users who want to access their saved Screening or Intake information must first create an account on a system called CentralID.
3. Users logging in to Universal Access with the Universal Access login screen can use their CentralID username/password to authenticate.
4. Users perform all of their account management using an external system we're calling CentralID (for example, resetting password, creating a new account, changing account details).
5. CentralID stores all user records in a secure LDAP server.
6. Because all account management is now performed in CentralID, the account creation screens and password reset screens are to be removed from Universal Access.
7. Users should be able to log in to Universal Access as soon as they have registered with CentralID, there should be no delay waiting for id to propagate to Universal Access.

All of these requirements are supported by the Universal Access External Security Integration. At the time of writing, addressing Single Sign On is beyond the scope of the External Security Integration – please contact Cúram Global Services for more information about how to support Single Sign On requirements.

Configuration Tasks

Taking the Analysis example, the following configuration tasks must be undertaken:

1. Configure the Application Server to use LDAP for authentication.
2. Deploy Cúram Universal Access in Identity Only mode for registered users.
3. Configure Cúram Universal Access so that Create Account screens are not displayed.
4. Configure Cúram Universal Access so that users are directed to register with the External System.

Alternate login ID configuration

By default, to authenticate users at login, Cúram uses a fixed user name that cannot be changed after it is created. However, you have the option of configuring an alternate login ID that can be updated.

For information about configuring alternate login IDs, see the related links. If you configure an alternate login ID for a user name that is case-sensitive, then the alternative login ID is also case-sensitive.

It is necessary to complete the LDAP setup task that is listed in the "Configuration Tasks" on page 10 topic only if the user is in `Identity–Only` mode. In `Identity–Only` mode, it is necessary to match the login IDs that are stored in LDAP with the login IDs that are stored in the Cúram `ExtendedUsersInfo` table.

Configure the Application Server to use LDAP for Authentication

Please refer to the relevant Application Server documentation for information on how to configure your Application Server to use LDAP for authentication.

Deploy Cúram Universal Access in Identity Only mode for Registered Users

Add the following properties to AppServer.properties:

```
curam.security.check.identity.only=true
curam.security.user.registry.disabled.types=EXT_AUTO,EXT_GEN
```

Then create a file called citizenworkspace\pageplayer\CustomPagePlayer.properties as a classpath resource containing:

```
curam.citizenworkspace.enable.usertypes.for.temporary.users=true
public.user.type=EXT_AUTO
```

To re-configure the application server run:

```
appbuild configure
```

The `curam.security.check.identity.only` property ensures that application security is set to work in Identity Only mode. For more information about Identity Only authentication mode please refer to the `Cúram Deployment Guide for WebSphere` or Cúram Deployment Guide for WLS as appropriate. In Identity Only mode authentication only uses the internal user table to check for the existence of the user. The validation of the password is left to a subsequent module, either a JAAS module (Oracle WebLogic) or the User Registry (IBM® WebSphere®).

Take the example of a user, "johnsmith", who has been registered with the CentralID LDAP server. In order for John Smith to be able to use Cúram Universal Access, there must also be a "johnsmith" entry in the ExternalUser table. When John Smith logs in, his authentication request is passed to the Cúram JAAS Login Module. This checks that the user "johnsmith" exists in the Cúram ExternalUser table but does not check the password. The authentication then proceeds to the User Registry (WebSphere) or LDAP JAAS Module (WebLogic) where the username and password are checked against the contents of the CentralID LDAP server. For this to work correctly it is necessary to configure the application server with the connection details for the secure LDAP server.

The Identity Only configuration allows the application to defer to an external security system such as an LDAP-based directory service for the authentication of user credentials. This does not work for anonymous users of Universal Access however. When a user accesses the front page of Universal Access for the first time, they are automatically logged in as the "publiccitizen" user. If they subsequently choose to Screen themselves or perform an Intake Universal Access creates a new "generated" anonymous user. Each generated user is unique and this ensures that the data belonging to that user is kept confidential. Neither the publiccitizen nor the generated users are inserted into the LDAP directory so they cannot be authenticated using the Identity Only mechanism. This is the purpose of the following line of configuration:

```
curam.security.user.registry.disabled.types=EXT_AUTO,EXT_GEN
```

This line ensures that users with the user type EXT_AUTO (the publiccitizen) and EXT_GEN (generated users) are authenticated against Cúram's External User table. Once the server has been configured with the above configuration and started, perform the following configuration steps:

1. Log in as sysadmin.
2. Select Application Data -> Property Administration.
3. Select Category "Citizen Account - Configuration".
4. Set the property 'curam.citizenaccount.public.included.user' to the value EXT_AUTO.
5. Set the property 'curam.citizenaccount.anonymous.included.user' to the value EXT_GEN.
6. Set the property 'curam.citizenworkspace.enable.usertypes.for.temporary.users' to the value true.
7. Publish the property changes.

   One final configuration entry is required in order to ensure that Universal Access operates correctly with respect to authentication, this change can be made as follows.
8. Log in as sysadmin.
9. Select Application Data -> Property Administration.
10. Select Category "Infrastructure – Security parameters".
11. Set curam.custom.externalaccess.implementation to 'curam.citizenworkspace.security.impl.CitizenWorkspacePublicAccessSecurity'.
12. Publish the property changes.

Finally, logout and restart the server. This configuration task should be complete at this point.

Configure Cúram Universal Access so that Create Account Screens are not Displayed

In the example above requirement 4 indicates that all Account Management functions are to be handled by the external system, CentralID. These include creation of a new account and performing a password reset. By default, Universal Access provides screens for these functions. These screens must be configured out in order to meet requirement 4 above:

1. Log in as sysadmin.
2. Select Application Data -> Property Administration.
3. Select Category "Citizen Portal - Configuration".
4. Set the property 'curam.citizenworkspace.enable.account.creation' to "NO".
5. Publish the property changes.

The above steps remove references to Account Creation pages from Universal Access. The Login Screen still contains a link to a Universal Access page for changing passwords. In this example the team implementing want to retain this link but change it to launch a new browser window on the CentralID password reset page. This can be achieved as follows:

1. Log in as sysadmin.
2. Select Application Data -> Property Administration.
3. Select Category "Citizen Portal - Configuration".
4. Set the property 'curam.citizenworkspace.forgot.password.url' to something like "http://www.centralid.gov/resetpassword"
5. Publish the property changes.

In order to remove the reset password link altogether use the following steps:

1. Log in as sysadmin.
2. Select Application Data -> Property Administration.
3. Select Category "Citizen Portal - Configuration".
4. Set the property 'curam.citizenworkspace.display.forgot.password.link' to "NO".
5. Publish the property changes.

Configure Cúram Universal Access so that users are directed to register with an External System

Out of the Box Universal Access invites users to log in with the message: "Please enter your User Name and Password and click the Next button to continue." Replacing this message is a good way of directing the non-registered user towards the CentralID screen for registration. For example the message on the Logon screen could read something like:

```
"<p>If you are registered with CentralID enter your username
   and password to log in. To register go to
   <a href="http://www.centralid.gov/register"> The CentralID
   registration page.</a></p>"
```

The properties for controlling the Login Page message can be found in `<CURAM_DIR>/EJBServer/components/Data_Manager/Initial_Data/blob/prop/Logon.properties`

To customize the message displayed, follow the procedure in *Customizable Universal Access Page Content*.

Development Tasks

The configuration tasks described so far allow customers to fulfill the requirements listed in the example with the exception of requirement:

```
"7 - Users should be able to log in to Universal Access
   as soon as they have registered with CentralID, there
   should be no delay waiting for id to propagate to other
   systems".
```

In order to function correctly, Cúram Universal Access requires each user to have an entry in the ExternalUser table. The customer could build a batch process to import users from the LDAP directory into the Cúram ExternalUser table but this would not satisfy requirement 7 since the user must be able to register with CentralID and then immediately use Universal Access. Another option would be to build a web service or similar mechanism that would be invoked when a new user registers with CentralID. The implementation of the web service would create the appropriate entry in the ExternalUser table.

This document however, now describes a simpler option which is to override the default login behavior to create new accounts on-the-fly, after checking that the relevant entry exists in the LDAP server.

Overriding the default login behavior in Universal Access can be done by extending the curam.citizenworkspace.security.impl.AuthenticateWithPasswordStrategy class and overriding the authenticate() method. The code below outlines how to use the AuthenticateWithPasswordStrategy and other security APIs to meet the requirements described above:

```
public class CustomSecurityStrategy extends AuthenticateWithPasswordStrategy {
  @Inject
  private CitizenWorkspaceAccountManager cwAccountManager;
  ...
  @Override
  public String authenticate(final String username,
      final String password)
      throws AppException, InformationalException {
    final String retval = null;
    if (username.equals(PUBLIC_CITIZEN)) {
      return super.authenticate(username, password);
    }
    // Authenticate generated accounts as normal
    if (cwAccountManager.isGeneratedAccount(username)) {
      return super.authenticate(username, password);
    }
    // Check that the user exists in LDAP
    // This prevents hackers from registering a lot of bogus
    // accounts that exist in Curam but not in LDAP
    if (!isUserInLDAP(username)) {
      return SECURITYSTATUS.BADUSER;
    }
    // If there's no account for this user
    if (!cwAccountManager.hasAccount(username)) {
      createUserAccount(username);
    }
    return SECURITYSTATUS.LOGIN;
  }
  private void createUserAccount(final String username)
      throws AppException, InformationalException {
    final CreateAccountDetails newAcctDetails;
    ...
    cwAccountManager.createStandardAccount(newAcctDetails);
  }
}
```

The code above checks to see if the user logging in is the publiccitizen user or a generated account. In both of these cases, authentication logic is delegated to the default AuthenticateWithPasswordStrategy. In the case of a registered user, the Strategy checks the LDAP directory to ensure that the user exists there. If the user exists in the LDAP directory and does not exist yet in Cúram, then a new user account is created. Note, the custom code does not need to authenticate the user against LDAP since the authentication is handled by the User Registry in WebSphere or the LDAP JAAS Module in WebSphere. It is important to note that the password parameter of the authenticate() method is passed in clear text.

In order to install the CustomSecurityStrategy it must be bound in place of the Default Security Strategy. This can be done by using a Guice Module to bind the implementation:

```
public class CustomModule extends AbstractModule {
  @Override
  protected void configure() {
    binder().bind(SecurityStrategy.class).to(
      CustomSecurityStrategy.class);
  }
}
```

The CustomModule must be configured at startup. This can be achieved by adding a DMX file to the custom component as follows:

```
<CURAM_DIR>/EJBServer/custom/data/initial/MODULECLASSNAME.dmx

<?xml version="1.0" encoding="UTF-8"?>
<table name="MODULECLASSNAME">
    <column name="moduleClassName" type="text" />
    <row>
      <attribute name="moduleClassName">
        <value>gov.myorg.CustomModule</value>
      </attribute>
    </row>
</table>
```

Customizing Universal Access Triage

A description of the customization points for the triage process.

For information about configuring and administering triage, see the `Cúram Universal Access Guide`.

Available triage events

There are a number of events which are fired during the triage process. These events can be broken into two categories, persistence events and custom events. The persistence events are standard data access events fired by the persistence infrastructure. The custom event is something that has been added to allow custom processing when the user performs a particular action. Both of these events are outlined below.

Standard persistence events

On running the triage rule set, the results of the session are persisted to the `TriageResult` entity. This triggers the invocation of the pre and post insert events. For information about how to make use of the `PersistenceEvent` API, see "Events" in the `Persistence Cookbook`.

Triage Referral Event

The `curam.triage.impl.TriageEvents.ReferralEvent.referralEmailSent` event is fired immediately after a citizen refers themselves for a service using Universal Access. For more information, see the API Javadoc for ReferralEvent in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc`.

Customizing Universal Access Screening

Use the supported customization points to customize screening.

For information on setting up and configuring screening, see the `Cúram Universal Access Configuration Guide`.

How to Track the Volume, Quality, and Results of Screenings

The curam.citizenworkspace.impl.CWScreeningEvents class is used for access to the events fired around screening, this could typically be used for tracking the volume or results of screening for reporting purposes. For further details, refer to the API Javadoc for CWScreeningEvents. This can be found in `<CURAM_DIR>/EJBServer/components/CitizenWorkspace/doc`

How to Populate a Custom Screening Results Page

The `curam.citizenworkspace.security.impl.UserSession` API contains a performScreening operation. You can use this operation to facilitate the population of a custom Screening Results page.

The Screening Results page is displayed when an IEG screening script is executed. The operation executes the configured rule set for the selected screening type. The results of the screening, that is, the list of eligible and undecided programs, are stored against the user's session.

The screeningResultsForDisplay return type of the operation allows access to three objects. These objects contain the information that is required to populate either the default or custom Screening Results page.

The three objects are as follows:

ScreeningType
The screening type that the user selected.

List<Program>
A list of the programs that the user was screened for. The ScreeningResultsOrderingStrategy dictates the order of the programs listed.

Map<String, ProgramType>
A join.util.map that contains a mapping of strings to ProgramTypes where the string contains the unique reference for that ProgramType.

A sample usage is illustrated below:

```
UserSession userSession = userSessionDAO.get(sessionID);
ScreeningResultsForDisplay screeningResultsForDisplay =
    userSession.performScreening();
```

A sample interface definition is illustrated below:

```
/**
   * Executes the Screening rule set associated with the current screening IEG
   * script data. The return object, {@link ScreeningResultsForDisplay},
   * contains all of the information required to be displayed on the
   * Screening Results page.
   *
   * @return object containing the ordered screening results, the selected
   *        {@link ScreeningType} and a map of {@link ProgramType} records.
   *
   * @throws InformationalException
   *          Generic exception signature.
   * @throws AppException
   *          Generic exception signature.
   */
  ScreeningResultsForDisplay performScreening() throws InformationalException,
      AppException;
```

For further details, refer to the API Javadoc for the `curam.citizenworkspace.security.impl.UserSession`. This can be found in `<CURAM_DIR>/EJBServer/components/CitizenWorkspace/doc`.

Customizing Application Intake Processing

Use these customization points to customize the application intake process up to the point of submitting an intake application.

For more information about configuring an intake application, see the `Cúram Universal Access Guide`.

How to Pre-populate the Intake Script

The `StartIntakeEvents.startIntake` is raised before an intake script is executed. This can be used to edit the contents of the data store before the intake process begins. Typically this will be done where, for example, a citizen has gone through screening and added some basic data as part of that process. This customization point allows for the transfer of this basic data to the intake. Note that the signature supplies a link to the datastore for pre-population. For further details, refer to the API Javadoc for `StartIntakeEvents`. This can be found in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc`

How to Add a Validation for Program Selection

An event is raised when processing the data entered by the user on the Select Intake Program screen. The event, curam.citizenworkspace.impl.ProgramSelectionEvents.intakeProgramsSelected, allows the validation of the programs selected by the user. This can be used to further apply business rules to an intake application for a citizen, where product combinations cannot be combined, for example. For further details, refer to the API Javadoc for `ProgramSelectionEvents`. This can be found in `<CURAM_DIR>/EJBServer/components/CitizenWorkspace/doc`

Customizing the Handling of Submitted Applications

Use these customization points to customized the application intake process after an intake application is submitted.

For more information about configuring an intake application, see the `Cúram Universal Access Guide`.

How to Customize the Process Intake Application Workflow

The Intake Application Workflow is summarized in the figure below.

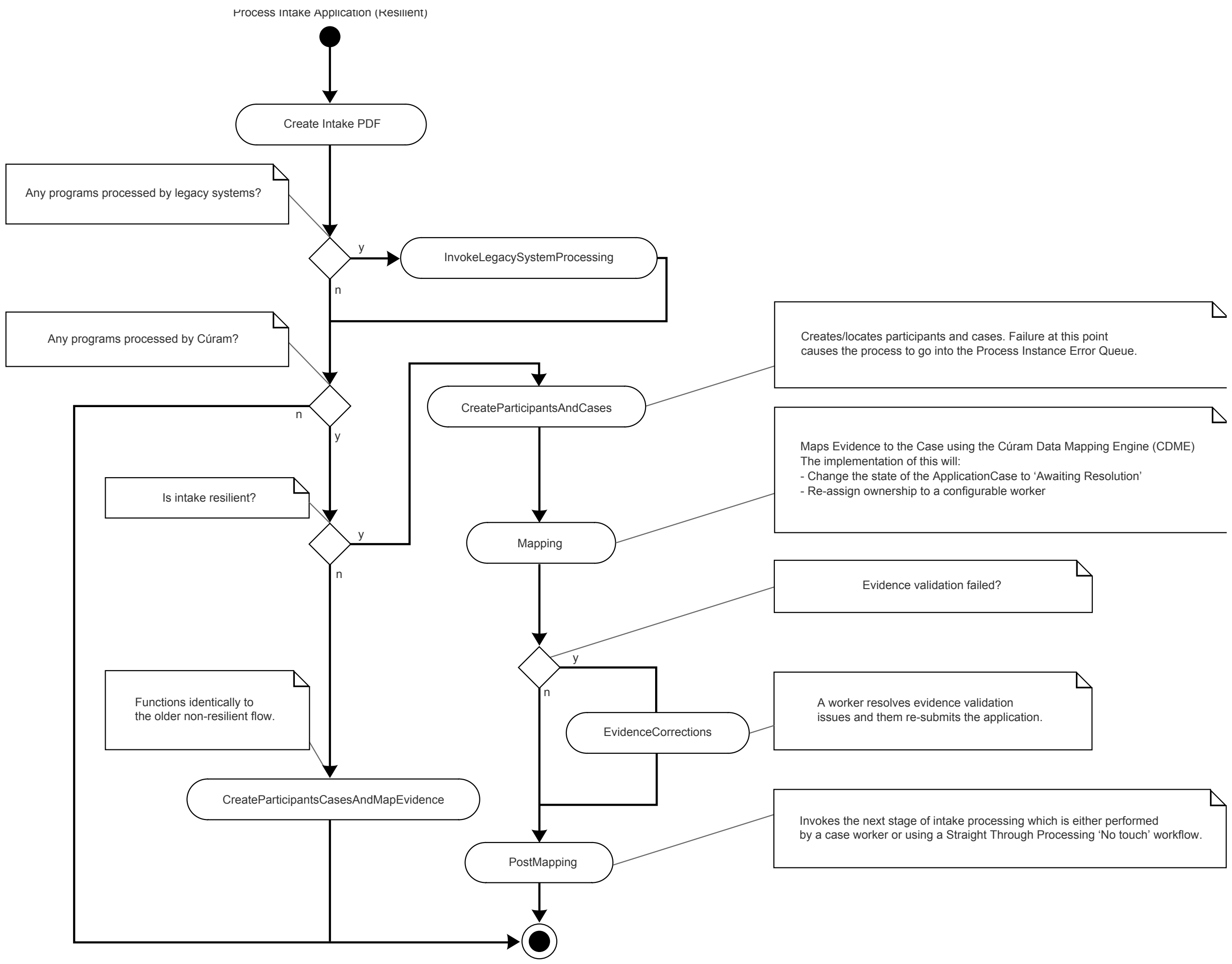
Process Intake Application (Resilient)
Create Intake PDF
Any programs processed by legacy systems?
y
n
InvokeLegacySystemProcessing
Any programs processed by Cúram?
n
y
CreateParticipantsAndCases
Creates/locates participants and cases. Failure at this point causes the process to go into the Process Instance Error Queue.
Maps Evidence to the Case using the Cúram Data Mapping Engine (CDME) The implementation of this will:
- Change the state of the ApplicationCase to 'Awaiting Resolution'
- Re-assign ownership to a configurable worker
Is intake resilient?
y
n
Mapping
Evidence validation failed?
y
n
Functions identically to the older non-resilient flow.
EvidenceCorrections
A worker resolves evidence validation issues and them re-submits the application.
CreateParticipantsCasesAndMapEvidence
Invokes the next stage of intake processing which is either performed by a case worker or using a Straight Through Processing 'No touch' workflow.
PostMapping

Figure 1. Intake Application Workflow

Create Intake PDF
This automatic activity creates a PDF document based on the content of the application. Customizing the generated PDF is described in "How to Customize the Generic PDF for Processed Applications" on page 19.

InvokeLegacySystemProcessing
This automatic activity sends applications to legacy systems via Web Services. This path is taken only if there are legacy systems associated with at least one of the programs on the application.

CreateParticipantsAndCases
This automatic activity creates participants for the submitted application and then creates a case or cases for the various programs associated with the application. In most cases, an Application Case or Cases are created. This path is taken if the value of the configuration property `curam.intake.use.resilience` is set to true. For reasons of backward compatibility, this property is set to false by default, however it is strongly recommended that all production systems set this value to true. For more information on the implications of setting this value to true, see "How to Use Events to Extend Intake Application Processing" on page 20.

Mapping
This automatic activity uses the Cúram Data Mapping Engine (CDME) to

map data collected in the application script into Case Evidence. Under most circumstances this will proceed smoothly. In the event that a validation issue occurs with the mapped evidence, this activity will be automatically re-tried. During the re-try, if there is a single Application Case, the validations will be disabled and a WDO flag `IntakeCaseDetails.mappingValidInd` set to false.

EvidenceCorrections
This manual task is invoked if the Mapping activity failed due to a validation error (`IntakeCaseDetails.mappingValidInd` set to false). The assignment of this task is configurable. For more information, see Evidence Issues Ownership Strategy in the `Intake Configuration Guide`. The assigned caseworker or operator will resolve the evidence validation issues and then re-submit the application.

PostMapping
This automatic activity kicks off the next stage of application processing by invoking the event `IntakeApplication.IntakeApplicationEvents.postMapDataToCuram()`.

CreateParticipantsCasesAndMapEvidence
This path is followed when the configuration property `curam.intake.use.resilience` is set to false. This automatic activity behaves identically to the old, non-resilient workflow. It creates cases and participants and performs all evidence mapping in a single transaction. This makes the process less resilient in the event of a failure.

Customers are free to customize the workflow in the usual recommended manner as described in the `Cúram Development Compliancy Guide` and `Cúram Workflow Management System Guide`. Note that customers should not make any changes to the enactment structs used by these workflows.

How to Customize the Generic PDF for Processed Applications

Universal Access provides functionality to map all intake applications to a generic PDF that records the values of all the information entered by the user. This PDF is rendered by the Cúram XML Server. Customers can override the default formatting of the generic PDF as follows:

copy:

- *CURAM_DIR/EJBServer/components/Workspaceservices/Data_Manager/InitialData/XSLTEMPLATEINST.dmx*

to:

- *CURAM_DIR/EJBServer/components/custom/Data_Manager/InitialData*

Edit the project\config\datamanager_config.xml to replace the entry for:

- *CURAM_DIR/EJBServer/components/Workspaceservices/Data_Manager/InitialData/XSLTEMPLATEINST.dmx*

with an entry for:

- *CURAM_DIR/EJBServer/components/custom/Data_Manager/InitialData/XSLTEMPLATEINST.dmx*

copy:

- *CURAM_DIR/EJBServer/components/Workspaceservices/Data_Manager/InitialData/blob/WSXSLTEMPLATEINST001*

to the directory:

- *CURAM_DIR/EJBServer/components/custom/Data_Manager/InitialData/blob.*

Edit this file to suit the needs of the project.

How to Use Events to Extend Intake Application Processing

The interface `IntakeApplication.IntakeApplicationEvents` contains events that get fired after the client has submitted an intake application for processing. These events can be used to change the way that intake applications get handled, for example to supplement or replace the standard CDME mapping or to perform an action after an application has been sent to a remote system using web services. For further details, please refer to the API Javadoc information for `IntakeApplication.IntakeApplicationEvents`. This can be found in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc`.

The interface `IntakeProgramApplication.IntakeProgramApplicationEvents` contains events that are fired at key stages during the processing of an application for a particular program. For further details, please refer to the API Javadoc information for `IntakeProgramApplication.IntakeProgramApplicationEvents`. This can be found in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc`.

Note: As of Cúram 6.0.5.5, there is a change to the ordering of the `IntakeApplication.IntakeApplicationEvents.preMapDataToCuram()` and `postMapDataToCuram()` events within the Intake Application Workflow when the configuration property `curam.intake.use.resilience` is set to true. When this property is set to true, `IntakeApplication.IntakeApplicationEvents.preMapDataToCuram()` is called in the Mapping activity prior to Evidence mapping. This means that the`IntakeApplication.IntakeApplicationEvents.preMapDataToCuram()` event is called after the cases and participants have been created by the CreateParticipantsAndCases activity. Previous versions of the intake process invoked this event prior to participant and case creation. When the configuration property `curam.intake.use.resilience` is set to true, the `IntakeApplication.IntakeApplicationEvents.postMapDataToCuram()` event is fired in the PostMapping activity.

How to Customize the Concern Role Mapping Process

The `curam.workspaceservices.applicationprocessing.impl` package contains a ConcernRoleMappingStrategy API that provides a customization point into the online application process.

Use the ConcernRoleMappingStrategy API to implement custom behavior following the creation of each new concern role that is added to an application. For example, customers who have customized the prospect person entity might want to store information on that entity that cannot be mapped using the default CDME processing.

Enable the ConcernRoleMappingStrategy API

In the administration application, enable the ConcernRoleMappingStrategy API by setting the Enable Custom Concern Role Mapping property to true.

Procedure

1. Log in to the Cúram System Administration application as a user with system administration permissions.
2. Click **System Configurations**>**Application Data**>**Property Administration**.
3. In the **Application - Intake Settings** category.
4. Search for the property curam.intake.enableCustomConcernRoleMapping.
5. Edit the property to set its value to true.
6. Save the property.
7. Select the Publish action.

Use the ConcernRoleMappingStrategy API

When enabled, use the ConcernRoleMappingStrategy API to implement a strategy for mapping information to a custom concern role.

About this task

The curam.workspaceservices.applicationprocessing.impl package contains the ConcernRoleMappingStrategy API.

Procedure

1. Provide an implementation of the customization point.
2. Bind your custom implementation by creating or extending your custom mapping module as follows:
   ```
   package com.myorg.custom;
   class MyModule extends AbstractModule {
     @Override
     protected void configure() {

         bind(ConcernRoleMappingStrategy.class).to(
              MyCustomConcernRoleMapping.class);
   ```
3. Add your MyModule class to the ModuleClassName table using an appropriate DMX file if you have not already done so.

How to Send Applications to Remote Systems for Processing

The Citizen Workspace can be used to send applications to remote systems for processing using web services. An event `ReceiveApplicationEvents.receiveApplication` is raised before the application is sent to the remote system. This can be used to edit the contents of the data store used to gather application data before transmission. For further details, refer to the API Javadoc for `ReceiveApplicationEvents`. This can be found in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc`.

Customizing the Citizen Account

The Citizen Account is a facility within Cúram Universal Access (UA) that allows a linked UA user to log in to a secure area where they can screen and apply for programs. The client also can view information relevant to them, including individually tailored messages, system-wide announcements, updates on their payments, contact information for agency staff and Outreach campaigns that might be relevant to them. The Citizen Account also provides a framework for customers to build their own Citizen Account pages or override the existing pages.

For a full description of the default functions, and for more information about linked UA users, see the *Cúram Universal Access Guide*.

Citizen Account Technical Overview

Unlike the rest of the Universal Access (UA) application, the Citizen Account framework is defined in the Cúram User Interface Metadata (UIM). This feature means that customers can override existing pages, add their own, and customize the navigation of the framework as they can customize the caseworker application.

The Citizen Account is built on top of the user interface infrastructure. It uses only a subset of the user interface and navigation components that are offered by the infrastructure to achieve a simple, usable application that clients can understand and use without any specific training.

Linked UA users perform UA triage, screening, and the online application by using their account. The Citizen Account includes UIM pages that are a view onto the triage, screening, and the online application functions. These pages are not configurable or customizable: the functions that they offer are configurable by using administration as specified in the documentation for these areas. The UIM pages that are related to triage, screening, and the online application are not to be modified or overridden.

Citizen Account Security Considerations

Displaying sensitive data to clients over the web inherently is dangerous and security must be a primary concern when administrators develop Citizen Account customizations. All public-facing applications must undergo rigorous security analysis and testing before they are deployed. Users must contact IBM support to discuss unusual customizations that might have specific security issues.

For more information, see the topic, Securing Universal Access (UA).

Refer also to the procedure Requesting a Product Enhancment (RFE) for a description of how to make the enhancement request.

Permission to call the server facade methods that serve data to Citizen Account pages is managed by the standard authorization model. For more information, see the *Cúram Server Developer* documentation. In addition to the standard authorization checks, each facade method that is called by a Citizen Account page must complete the following security checks to ensure the user who is associated with the transaction (the currently logged in user) has permission to access the data they are requesting:

- Ensure that the currently logged in user is of the correct type. They must be an External user with an `applicationCode` of `CITWSAPP`, and have a UA account of type `Linked`.
- Ensure that the currently logged in user has permission to access the specific records that they are reading. For instance, validate any page parameters that are passed in to ensure that the records requested are related to the currently logged in user in some way.

Securing Universal Access

Use this information to understand the Universal Access Security Model and how to customize Universal Access securely in line with this model.

Cúram Server Developer

Use this information to learn about the server development environment, which enables the development of high-quality, low-cost client/server applications through model driven code generation. This generation facilitates client/server development by taking a UML model and producing generated Java code; a data definition language which describing the database entities in the model, and support for remote invocation.

Ensure That the Currently Logged In User is the Correct Type

The `curam.citizenaccount.security.impl.CitizenAccountSecurity` application programming interface (API) offers a method `performDefaultSecurityChecks` that ensures that the user is the correct type. This method checks the user type, and if not acceptable, writes a message to the logs and fails the transaction.

Note: This API needs to be called in the first line of every custom facade method before any processing or further validation takes place:

```
public CitizenPaymentInstDetailsList listCitizenPayments()
    throws AppException, InformationalException {

    // perform security checks
    citizenAccountSecurity.performDefaultSecurityChecks();

    // validate any page parameters (none in this case)

    // invoke business logic
    return citizenPayments.listPayments();
  }
```

Ensure the Currently Logged In User Has Access to the Specific Records They Requested

A malicious user who is logged in to a valid linked Cúram Universal Access (UA) account might send requests to the system that is requesting data related to other users. To prevent this intrusion from happening, all page parameters must be validated to ensure that they are somehow traceable back to the currently logged in user. How this conclusion is determined is different for each type of record.

For example, a **Payment** can be traced back to the **Participant** by way of the **Case** on which it was entered.

The `curam.citizenaccount.security.impl.CitizenAccountSecurity` application programming interface (API) offers methods to complete these checks for the types of records that are served to clients by the initially configured pages. For specific information, review the Javadoc of this API. For custom pages that serve different types of data, additional checks must be implemented to validate the page parameters.

This process needs to be added to a custom security API and called by the facade methods in question. The methods need to check to see whether the record requested can be traced back to the currently logged in user, and if not, it needs to log the user name, method name, and other data. If these conditions are not met, the transaction needs to be failed immediately (as opposed to adding the issue to the validation helper and allowing the transaction to proceed):

```
if (paymentInstrument.getConcernRole().getID()
   != citizenWorkspaceAccountManager
     .getLoggedInUserConcernRoleID().getID()) {

 /**
 * the payment instrument passed in is not related
 * to the logged in user log the user name of the
 * current user, the method invoked and any other
 * pertinent data
 */
 // throw a generic message
 throw PUBLICUSERSECURITYExceptionCreator
   .ERR_CITIZEN_WORKSPACE_UNAUTHORISED_METHOD_INVOKATION();
 }
```

While as much information as possible regarding the infraction needs to be logged, it is important to ensure that the exceptions thrown do not display any information that might be useful to malicious users. A generic exception needs to be thrown that does not contain any information that relates to what went wrong. The `curam.citizenaccount.security.impl.CitizenAccountSecurity` API throws a generic message that states `You are not privileged to access this page`.

How to Add a New Page to Citizen Account

Use the information that is provided to learn how to add a page to a Citizen Account. The links that are provided show how to complete the tasks that are required to add a customized page to the account.

The following tasks need to be carried out to add a custom page to a Citizen Account.

Create a Custom, External Client Component

Artifacts that form part of public facing applications, such as Cúram Universal Access (UA), need to be stored in separate components to avoid deploying internal pages intended for administrators or caseworkers into public facing applications. Use this information to understand which artifacts need to be included in the customized page and which components not to include in the page.

The first step in adding a custom page to citizen account is to set up a new custom client-side component in which to put your page. For instructions on how to complete this task, refer to External Client Applications in the *IBM Cúram Server Developer* documentation. This component needs to include the artifacts to be deployed to UA, and omit any artifacts intended for use by administrators or caseworkers. This component must be added to the deployment packaging for the UA Enterprise ARchive (EAR) file. The important point is that the `deployment_packaging.xml` file that is included in the `<CURAM_DIR>/EJBServer/project/config/` folder contains the minimum entries for the components that need to be built, specifically `CitizenAccount`, `CitizenWorkspace`, `IntelligentEvidenceGathering`, and `CEFWidgets`.

External Client Applications:
Due to the webclient directory containing a mix of components that are targeted for different EAR packaging, it can be difficult to use the single development environment and component order to develop and test these.

To allow for this a build target`external-client` will allow for creation of an environment and building of the components specified for an EAR entry in the `deployment_packaging.xml`.

The target requires a parameter`-Dapp` which should refer to the name of an EAR entry within the `deployment_packaging.xml`.

```
build external-client -Dapp=SamplePublicAccess
```

Figure 2. external-client invocation

The build target will copy the components specified for this EAR entry to a `webclient\build\apps\<app name>` directory and here will both build the project and create the relevant Eclipse project configuration files to allow for the project directory to be imported into Eclipse and development-type testing to be performed on these external client applications.

Citizen Account Security Considerations:

Displaying sensitive data to clients over the web inherently is dangerous and security must be a primary concern when administrators develop Citizen Account customizations. All public-facing applications must undergo rigorous security analysis and testing before they are deployed. Users must contact IBM support to discuss unusual customizations that might have specific security issues.

For more information, see the topic, Securing Universal Access (UA).

Refer also to the procedure Requesting a Product Enhancment (RFE) for a description of how to make the enhancement request.

Permission to call the server facade methods that serve data to Citizen Account pages is managed by the standard authorization model. For more information, see the *Cúram Server Developer* documentation. In addition to the standard authorization checks, each facade method that is called by a Citizen Account page must complete the following security checks to ensure the user who is associated with the transaction (the currently logged in user) has permission to access the data they are requesting:

- Ensure that the currently logged in user is of the correct type. They must be an External user with an `applicationCode` of `CITWSAPP`, and have a UA account of type `Linked`.
- Ensure that the currently logged in user has permission to access the specific records that they are reading. For instance, validate any page parameters that are passed in to ensure that the records requested are related to the currently logged in user in some way.

Create a UIM page in the new component

Develop the custom Cúram User Interface Metadata (UIM) page in the new client component. The user interface infrastructure offers a wide array of complex functions. Bear in mind that the target audience of this page is clients who are not familiar with complex user interfaces. Therefore, it is advisable to keep Citizen Account pages relatively simple, compared with the complex user interfaces developed for experienced user types such as caseworkers or administrators.

While UIM screens can be used in the Universal Access user interface, only a sub set of UIM features is fully supported by Universal Access. The following table summarizes the UIM elements that are supported. The first column lists the UIM elements. The next three columns represent the main contexts that a feature can be displayed in. Taking the first row as an example:

`ACTION_SET - PAGE`

This coding refers to the use of an `ACTION_SET` as a child of the `PAGE` element. It is supported in both the **Main Content** pane and in a modal dialog. However, it is not supported within a pane in an expandable list. The **Notes** column provides additional information where necessary.

Table 12. Supported UIM features

UIM Element	Main Content Pane	MODAL	DETAILS_ROW	Notes
`ACTION SET - PAGE`	Up to 2 items, which are divided by a separator	Supported	Unsupported	Buttons are given a new style
`ACTION SET - CLUSTER`	Supported	Supported	Unsupported	Buttons are given a new style
`ACTION SET [@TYPE="LIST_ROW_MENU"] - LIST`	Unsupported	Unsupported	Unsupported	Generally not needed for Universal Access (UA). **Note:** An `ACTION_CONTROL` can be used in the column of a list.
`ACTION SET - LIST`	Unsupported	Unsupported	Unsupported	
`ACTION CONTROL - CONTAINER`	Supported	Supported	Supported	
`PAGE TITLE`	Supported	Supported	n/a	
`DESCRIPTION`	Supported	Supported	n/a	
`CLUSTER`	Supported	Supported	Supported	The collapsible behavior is not supported. Nesting of clusters is not supported.
`LIST`	Supported	Supported	Supported	The collapsible behavior is not supported. Paginated lists, scrollable lists, and nested lists are not supported.
`DETAILS ROW - LIST`	Supported	Supported	Supported	The `DETAILS_ROW` element refers to the feature known as Expandable Lists. Nesting of expandable lists is out of scope.
`FIELD`	Text input, Text area, codetable drop downs, date picker, password are all in scope	Text input, Text area, codetable drop downs, date picker, password are all in scope	Unsupported	Codetable hierarchy and all other items that are not defined in the cell for Main Content or Modals are not supported

Add Navigation for the New Page

The user can navigate to the new page either by using the main Cúram Universal Access (UA) **navigation** menu or using the **banner** menus.

Using the Universal Access navigation menu to access a new page

This is done in the standard way. for more information, see the *Cúram User Interface Developers Guide* for information regarding how to extend navigation.

```
<nc:navigation id="CitizenAccount">
    <nc:nodes>
      <nc:navigation-page
        id="home" page-id="CitizenAccount_certification"
          title="leaf.title.certification" />
    </nc:nodes>
  </nc:navigation>
```

Figure 3. Sample custom navigation entry for custom citizen account page

```
<nc:navigation id="StandardUser">
  <nc:nodes>
    <nc:navigation-page
      id="home" page-id="StandardAccount_certification"
      title="leaf.title.certification" />
  </nc:nodes>
</nc:navigation>
```

Figure 4. Sample custom navigation entry for custom standard account page

Using banner menus to access a new page

In addition to adding to the navigation menus, new pages can be incorporated into banner menus including the Mega-Menu which, in the initially configured application, is titled **Find Government and Community Help**. For more information, see the Navigation Configuration section.

Navigation Configuration section.

Create a Facade

Develop a facade that the page can call. This facade retrieves data based on either the currently logged in user, or page parameters that are passed in. Generally, Citizen Account pages read data that is related to the logged in user's linked accounts. Specifically, if the logged in user is linked to a Cúram participant, that is, a `concernRoleID`, then data that relates to that concern role, their cases, and their evidence is played. If the user is linked to remote case processing systems, then data from those remote systems can be displayed in the Citizen Account.

The `curam.citizenworkspace.security.impl.CitizenWorkspaceAccountManager` application programming interface (API) offers a convenience method that can be used to retrieve the linked identities of a currently logged in user that includes their linked `ConcernRole` if they have one. It is recommended that customers use this API to retrieve linked identities, as it has embedded security checks to ensure that the user in question is in fact a linked Cúram Universal Access (UA) user.

Relevant authorization entries must be added to the correct security groups to grant access to the new facade. If the page is to be accessed by users with a Standard Account, then the following steps must be taken. In the directory

`<CURAM_DIR>/EJBServer/component/custom/data/initial`, create a `NORMALSECURITYGROUP.dmx` and add the following entry to this file:

```
<table name="SECURITYGROUPSID">
  <column name="groupName" type="text" />
  <column name="sidName" type="text" />
  <column name="lastWritten" type="timestamp" />
  <row>
    <attribute name="groupName">
      <value>NORMALCITIZENWORKSPACEGROUP</value>
    </attribute>
    <attribute name="sidName">
      <value>MyCustomFacadeName.myCustomFacadeMethodName</value>
    </attribute>
    <attribute name="lastWritten">
      <value />
    </attribute>
  </row>
  ...
</table>
```

Similarly, if the facade is to be accessed by a user with a Linked Citizen Account, then create a `LINKEDSECURITYGROUP.dmx` entry under `<CURAM_DIR>/EJBServer/component/custom/data/initial` and add the following entry:

```
<table name="SECURITYGROUPSID">
  <column name="groupName" type="text" />
  <column name="sidName" type="text" />
  <column name="lastWritten" type="timestamp" />
  <row>
    <attribute name="groupName">
      <value>LINKEDCITIZENWORKSPACEGROUP</value>
    </attribute>
    <attribute name="sidName">
      <value>MyCustomFacadeName.myCustomFacadeMethodName</value>
    </attribute>
    <attribute name="lastWritten">
      <value />
    </attribute>
  </row>
  ...
</table>
```

How to Customize Universal Access Stylesheets in a Citizen Account

Citizen Account Cúram Universal Access (UA) stylesheets are customizable in the standard way for Cúram User Interface Metadata (UIM) pages. For more information, see the `Curam User Interface Developers` documentation. The Citizen Account stylesheets are in `webclient/components/CitizenAccount/css`.

Customizing Locale

The method for adding new locales to a Citizen Account is the same as for standard Cúram User Interface Metadata (UIM) pages as opposed to the dynamic manner by which public Cúram Universal Access (UA) pages can be globalized. To add different locales to a Citizen Account, the client project must be rebuilt to generate the JavaServer Pages (JSPs) in the new locale.

For more information, see the *Cúram Web Client Developer* documentation on managing UIM pages that are to be offered in multiple locales. The `CT_APPLICATION_CODE` codetable is used to map External User application codes to the specific UIM page they need to be routed to when they log in. The client infrastructure uses these configurations to determine where the user needs to be routed following a successful authentication. UA includes an entry for its default locale of `en`:

```
<code default="false" java_identifier="CITIZEN_WORKSPACE"
 status="ENABLED" value="CITWSAPP">
  <locale language="en" sort_order="0">
    <description>CitizenAccount_home</description>
  </locale>
</code>
```

When additional locales are added to UA, more entries must be added to this codetable for each locale that is being added. All entries need to contain the same description, which contains the value of the Citizen Account home page. This requirement also is true of other codetables that are used by UA, such as `CT_SecretQuestionType`. The names of the secret questions must be added in each locale that UA is to be offered in.

Citizen Account home page

The Citizen Account home page offers a range of functions to offer pertinent information to the citizen. Refer to the IBM Cúram Universal Access documentation. The various areas are configured in different ways, which are outlined in the Cúram Universal Access Configuration documentation.

When instructions refer to customizing the Citizen Account home page, references pertain to the constituent parts of the page, the **Outreach (Did You Know?)**, and **My Messages Panels**.

Customizing Display Text

The welcome message, cluster titles, and text of the last logged on message are stored in a properties file in the resource store. The logged in user name is appended to the property `citizenaccount.welcome.caption` property to display the welcome message on the home page. To change the welcome message, this property needs to be changed in the properties file.

The file is at `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageMyPayments.properties`. The text can be customized by uploading a new version to the resource store with a name attribute of `CitizenAccountHome`.

Outreach Campaigns

Outreach campaigns display targeted campaign messages to the client. These messages can include images and links to both Cúram Universal Access (UA) pages and to external websites. Whether a specific campaign should be displayed to a particular client is determined by a Cúram Express Rules (CER) rule set that is associated with the campaign in **Administration**. Outreach is developed using Advisor and CER, and the campaign output is recorded as **Advice Items**.

For more information, see the Cúram Universal Access Configuration documentation.

How to Configure a New Citizen Campaign:

for more information, see the `Universal Access Configuration` documentation on how to configure new Outreach campaigns in Administration.

Configuring Universal Access:

Various configuration options are available for IBM Cúram Universal Access. These are listed below.

Outreach Campaign Rule Sets:

Outreach is built on top of Advisor infrastructure, and the `CoreCitizenCampaignRuleset`, which all Outreach campaigns need to extend in turn inherits from the `CoreAdvisorRuleset`. The `CoreCitizenCampaignRuleset` is at `/EJBServer/components/citizenworkspace/CREOLE_Rule_Sets`.

The `CoreCitizenCampaignRuleset` defines two rule classes that are used to drive Outreach campaigns:

`CitizenCampaignAdmin` rule class

This rule class is a Cúram Express Rules (CER) rules representation of a `CitizenCampaign` administration record. The name, expiry date time, and image reference for a campaign are propagated. A rule object of this class exists for each active Outreach campaign in the system. These objects are managed internally by the Outreach infrastructure.

`AbstractCampaignAdviceItem` rule class

This class extends `AbstractAdviceItem` (see Advisor documentation). This rule class is the set that concrete Outreach campaign rule classes must extend. Concrete Outreach campaign rules classes must specify the following attributes that are inherited form this rule class:

- `citizenCampaignName`

  Names are unique across campaigns as they are used as a unique identifier.

  Note: The `citizenCampaignName` specified in the rule set and the name of the Outreach campaign when it is created in Administration must be identical. Therefore, when the new Outreach campaign is created in Administration, the name of the new campaign must match the `citizenCampaignName` specified in its associated rule set.
- `campaignShowAdvice`

  This attribute is where the campaign business logic lives. It returns `true` if the participant in question meets the criteria to display the campaign.

The `AbstractCampaignAdviceItem` class sets the `showAdvice` attribute of its parent based on whether a `CitizenCampaignAdmin` rule object exists for the campaign in question. For example, is the campaign active in Administration) and based on the value of the `campaignShowAdvice` attribute.

By default, the expiry date time of a campaign is taken from the Outreach campaign administration record. This information allows administrators to configure the expiry of campaigns. However, it also is possible to determine the expiry date time based on business logic or other rules if they want, by overriding the `expiryDateTime` attribute of the `AbstractCampaignAdviceItem` class in their child implementation of this class.

Concrete campaign rule classes must also declare a class that extends the Advisor `AbstractAdviceContext`. For more information, see the *Advisor* documentation and the following sample campaign rule set.

```
<RuleSet name="SampleCampaignRuleSet">

 <!-- This class is infrastructure used by Advisor,
   please refer to the Advisor documentation for more
   information. -->

 <Class extends="AbstractAdviceContext"
   extendsRuleSet="CoreAdvisorRuleSet"
     name="SampleCampaignContext">

   <!-- populated by advisor propagator -->
   <Attribute name="concernRoleID">
     <type>
       <ruleclass name="NumberParameter"
         ruleset="CoreAdvisorRuleSet"/>
     </type>
     <derivation>
       <specified/>
     </derivation>
   </Attribute>

   <!-- populated by advisor propagator -->
   <Attribute name="adviceContextID">
     <type>
       <javaclass name="Number"/>
     </type>
     <derivation>
       <specified/>
     </derivation>
   </Attribute>

   <Attribute name="advice">
     <type>
       <javaclass name="List">
         <ruleclass name="AbstractCampaignAdviceItem"
           ruleset="CoreCitizenCampaignRuleset"/>
       </javaclass>
     </type>
     <derivation>
       <fixedlist>
         <listof>
           <ruleclass name="AbstractCampaignAdviceItem"
             ruleset="CoreCitizenCampaignRuleset"/>
         </listof>
         <members>
         <!-- This list of members must include the custom rule
                class that extends AbstractCampaignAdviceItem -->
           <create ruleclass="SampleCampaign">
             <this/>
           </create>
         </members>
       </fixedlist>
     </derivation>
   </Attribute>
 </Class>

 <!-- Concrete Campaign / Advisor class that extends
         AbstractCampaignAdviceItem -->
 <Class extends="AbstractCampaignAdviceItem"
   extendsRuleSet="CoreCitizenCampaignRuleset"
     name="SampleCampaign">

   <!-- initialise the Advisor context. Please see Advisor
```

```
          documentation for more information -->
    <Initialization>
      <Attribute name="sampleCampaignContext">
        <type>
          <ruleclass name="SampleCampaignContext"/>
        </type>
      </Attribute>
    </Initialization>

    <!-- This is a reference to the campaign text stored in the
            resource store. Please see the Advisor documentation
            for more information. -->
    <Attribute name="adviceText">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <String value="propertyName"/>
      </derivation>
    </Attribute>

    <!-- This is a reference to the advice context ID.
         Please see the Advisor documentation for more
         information. -->
    <Attribute name="adviceContext">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <reference attribute="adviceContextID">
          <reference attribute="sampleCampaignContext"/>
        </reference>
      </derivation>
    </Attribute>

    <!-- This is used by the parent abstract class to read the
         campaign rule object. This name must be identical to
         the name given to the Outreach campaign in
         Administration -->
    <Attribute name="citizenCampaignName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <String value="SampleCampaign"/>
      </derivation>
    </Attribute>

    <!-- Whether or not to display the campaign for the given
            participant (provided the campaign is Active) -->
    <Attribute name="campaignShowAdvice">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <!-- business logic for campaign goes here. -->
        <true/>
      </derivation>
    </Attribute>
  </Class>
</RuleSet>
```

Images and Links:

Advisor and Outreach support including images and links as part of an Advice Item/campaign. The image itself is uploaded when a new Outreach campaign is created. By default, if an image is specified when the campaign is created in Administration, it is displayed without a link as part of the campaign. However, it is possible to specify a link within the rule set, and within that link to specify an image, referencing the image that was configured for the campaign.

Within the custom concrete campaign rule set, define a link:

```
<Class extends="AbstractLink"
   extendsRuleSet="CoreAdvisorRuleSet"
     name="ChildCareOptionLinkWithImage">

   <Attribute name="name">
     <type>
       <javaclass name="String"/>
     </type>
     <derivation>
       <String value="childCareOptionLinkImage"/>
     </derivation>
   </Attribute>

   <Attribute name="target">
     <type>
       <javaclass name="String"/>
     </type>
     <derivation>
       <String value="http://www.yourtargeturl.com"/>
     </derivation>
   </Attribute>

   <Attribute name="modal">
     <type>
       <javaclass name="Boolean"/>
     </type>
     <derivation>
       <false/>
     </derivation>
   </Attribute>

   <Attribute name="external">
     <type>
       <javaclass name="Boolean"/>
     </type>
     <derivation>
       <true/>
     </derivation>
   </Attribute>

   <Attribute name="linkImage">
     <type>
       <ruleclass name="Image" ruleset="CoreAdvisorRuleSet"/>
     </type>
     <derivation>
     <!-- note that this is specified. The parent rule class will
             specify the image reference from the campaign -->
       <specified/>
     </derivation>
   </Attribute>

 </Class>
```

When this link is declared within the custom implementation of `AbstractCampaignAdviceItem`, you specify the reference to the image configured in administration: For more information on defining links and images, see the *Advisor* documentation.

```
<Attribute name="childCareOptionLinkWithImage">
   <type>
     <ruleclass name="ChildCareOptionLinkWithImage"/>
   </type>
   <derivation>
     <create ruleclass="ChildCareOptionLinkWithImage">
       <specify attribute="linkImage">
         <reference attribute="campaignImage"/>
       </specify>
     </create>
   </derivation>
 </Attribute>
```

For the links related to image-only campaigns to be persisted to the Advisor, database tables (and displayed in Outreach Campaigns), an entry in the properties file related to that campaign is required. For example:

```
AdviceItem.imageOnlyText={link::imageCampaignLinkWithImage}
```

This entry does not designate a name for the link, but refers to the name of the rules object defined in the campaign rule set to represent this link.

Performance Considerations:

Campaigns refer to Cúram Express Rules (CER) rule objects to determine whether to display campaigns. Therefore, when the underlying data that these rule objects depend on changes, CER reassessment is triggered. This change causes Advisor to recalculate whether the campaign needs to be displayed. This change might affect performance and needs to be considered. Two types of data changes are involved:

Changes to the Participants' data

These kinds of changes affect a specific participant. For example, a campaign that references a client's address. Every time the user changes their address, this change would be propagated to the rule object that represents that participants' address. Because the campaign rule object depends on this operation, reassessment would be triggered. This reassessment means that every time the participant changes their address, the campaign rules is run to determine whether it still is displayed. Therefore, it is important to consider how often a piece of data might change, and for how many clients. Also, it needs to be considered whether refereeing to it in a campaign might cause a performance issue within the system.

Changes to Outreach Campaigns in Administration

These kinds of changes affect all the rule executions related to the campaign in question. This change triggers reassessment for every client who was assessed for eligibility for this campaign. For example, if the image associated with a campaign is changed, the system reexecutes the rules for each client that was considered for this campaign. This action might require a significant amount of processing that might have a performance impact on the system. It is encouraged that changes in campaign administration are undertaken when the system is not under a heavy load, or after the system is taken offline for maintenance.

My Messages

When a linked client logs in, messages are gathered for display from around the system, and from remote systems. This work is done by the `curam.citizenmessages.impl.CitizenMessageController` application programming interface (API). It reads persisted messages by participant from the `ParticipantMessage` database table, and also raises theCitizenMessagesEvent.userRequestsMessages event, inviting listeners to add messages to a list it passes as part of the event parameter. The messages that are gathered from each source are sorted, turned into XML and returned to the client for display.

Refer to the *Cúram Universal Access* documentation for an overview of the functions offered by the **Messages** pane and the specific messages offered in the initial configuration.

Configuring Citizen Messages:

Global configurations are included that can be specified for Citizen Messages, such as enabling certain types and configuring their display order. The different types of messages also include their own configuration points. Specific information about how to customize the various message types is provided later.

For more information on how to change the global configurations and on delivering Citizen Messages that use web services, see the *IBM Cúram Universal Access Configuration* documentation.

The textual content of a message type also can be configured. Each message type has a related properties file that includes the localizable text entries for the various messages displayed for that type. These properties also include placeholders that are substituted for real values related to the client at run time.

The wording of this text can be customized, by inserting a different version of the properties file into the resource store. The following table defines which properties file need to be changed for each type of message:

Table 13. Message properties files

Message type	Property file name
Payments	`CitizenMessageMyPayments.properties`
Application Acknowledgment	`CitizenMessageApplicationAcknowledgement.properties`
Verifications	`CitizenMessageVerificationMessages.properties`
Meetings	`CitizenMessageMeetingMessages.properties`
Referral	`CitizenMessagesReferral.properties`
Service Delivery	`CitizenMessagesServiceDelivery.properties`

It is also possible to remove placeholders (which are populated with live data at runtime) from the properties. However, there is currently no means to add further placeholders to existing messages. A custom type of message must be implemented in this situation.

Adding a New Type of Citizen Message:

Messages are gathered by the controller in two ways: the controller reads messages that were persisted to the database by using the `curam.citizenmessages.persistence.impl.ParticipantMessage` application programming interface (API), and also gathers them by raising the `curam.participantmessages.events.impl.CitizenMessagesEvent`

A decision needs to be made regarding whether to 'push' the messages to the database, or else have them generated dynamically by a listener that listens for the event that is raised when the client logs in. The specific requirements of the message type need to be considered, along with the benefits and drawbacks of each option.

Persisted Messages

In this scenario, when something takes place in the system that might be of interest to the client, a message is persisted to the database. For example, when a meeting invitation is created, an event is fired. The initially configured meeting messages function listens for this event. If the meeting invitee is a participant with a linked Cúram Universal Access (UA) account, a message is written to the `ParticipantMessage` table that informs the client that they are invited to the meeting.

One benefit of this approach is that little processing is done when the client logs in to see this message: the message is read from the database and displayed, as opposed to calculation that takes place that would determine whether the message was required. However, the implementation also needs to handle any changes to the underlying data that might invalidate or change the message, and take appropriate action.

For example, the meeting message function also listens for changes to meetings to ensure the meeting time, location, and similar, are up to date, and to send a new message to the client to inform the client that the location or time was changed.

Dynamic Messages

These messages are generated when the client logs in, by event listeners that listen for the `curam.participantmessages.events.impl.CitizenMessagesEvent.userRequestsMessages` event.

A benefit is that because the message is generated at runtime, code is not required to manage change over time. The message is generated based on the data within the system each time the client logs in. If some underlying data changes, the next time the client logs in, they will get the correct message.

A drawback to this approach is that significant processing might be required at run time to generate the message. Care must be taken to ensure that this processing does not adversely affect the load time of the Citizen Account home page.

Performance considerations must be evaluated against the requirements of the specific message type and the effort that is required to manage change to the data that the message is related to over time. For example, the initially configured verification message is dynamic. When a client logs in, it checks to see whether any outstanding verifications exist for that client. This process is a relatively simple database read, whereas it would be complicated to listen for various events in the Verification Engine and ensure that an up-to-date message was stored in the database related to the participants' outstanding verifications. Alternatively, , the meeting messages need to inform the client of changes to their meetings, so functionality had to be written to manage changes to the meeting record and its related message over time.

Implementing a new message type: In order to implement a new message type, regardless of whether the message will be persisted or generated dynamically, the following must be done:

Common Tasks

- Add a new entry to the `CT_ParticipantMessageType` codetable to represent the new message type. This will be used in administration to configure the new message type.
- Add a DMX entry for the ParticipantMessageConfig database table. This will store the type and sort order of the new message type and is used for administration. For example:

```
<row>
   <attribute name="PARTICIPANTMESSAGECONFIGID">
     <value>2110</value>
   </attribute>
   <attribute name="PARTICIPANTMESSAGETYPE">
     <value>PMT2001</value>
   </attribute>
   <attribute name="ENABLEDIND">
     <value>1</value>
   </attribute>
   <attribute name="SORTORDER">
     <value>5</value>
   </attribute>
   <attribute name="VERSIONNO">
     <value>1</value>
   </attribute>
 </row>
```

- Add a properties file to the App Resource store that contains the text properties and image reference for the message.
- Add an image for this message type to the resource store.

Implementing a dynamic message

In order to implement a dynamic style message, an event listener needs to be implemented, to listen for the `CitizenMessagesEvent.userRequestsMessages` event. This event argument contains a reference to the Participant and a list, to which the listener will add `curam.participantmessages.impl.ParticipantMessage` Java objects. For further details please consult the Javadoc API for `CitizenMessagesEvent`. This can be found in `<CURAM_DIR>/EJBServer/components/core/doc`

Developers should also refer to the Javadoc API for `curam.participantmessages.impl.ParticipantMessage` and `curam.participantmessages.impl.ParticipantMessages` for a full explanation.

The message text is stored in a properties file in the resource store. A dynamic listener will retrieve the relevant properties from the resource store, and create the ParticipantMessage object accordingly. The message text for a given message can include placeholders. Values for placeholders are added to ParticipantMessage objects as parameters. The CitizenMessagesController will resolve these placeholders, replacing them with the real values related to the participant in question that have been added as parameters to the message object.

Take for example this entry from the CitizenMessageMyPayment.properties file:

```
Message.First.Payment=
  Your next payment is due on {Payment.Due.Date}
```

The actual payment due date of the payment in question will be added to the ParticipantMessage object as a parameter (see example code below). The CitizenMessagesController then resolves the placeholders, populating the text with real values, and then turns the message into XML that is rendered on the citizen account home page (there is also a public CitizenMessageController method that will return all messages for a citizen as a list, please see the Javadoc information).

From `curam.participantmessages.impl.ParticipantMessage API`:

```
/**
 * Adds a parameter to the map. The paramReference
 * should be present in the message title or body so
 * it can be replaced by the paramValue before the message
 * is displayed.
 *
 * @param paramReference
 * a string place holder that is present in either the
 * message title or body. Used to indicate where the value
 * parameter should be positioned in a message.

 * @param paramValue
 * the value to be substituted in place of the place holder
 */
 public void addParameter(final String paramReference,
   final String paramValue) {

   parameters.put(paramReference, paramValue);
 }
```

The call to the method would look like this:

```
participantMessage.addParameter("Payment.Due.Date", "1/1/2011");
```

Messages can also include links. Similarly to placeholders, links are resolved at runtime. Links can use placeholder values as the text to be displayed for that link. A link is defined in a properties file as such:

```
Click {link:here:paymentDetails} to view the payment details.
```

In this example, "here" is the text to display, and "paymentDetails" refers to the name of the link that is to be inserted at that point in the text. Please see the `Advisor Developer's Guide` for more information. In order for a dynamic listener to populate this link with a target, it would create a `curam.participantmessages.impl.ParticipantMessageLink` object, specifying a target and a name for the link. The code would look like this:

```
ParicipantMessageLink participantMessageLink =
     new ParticipantMessageLink(false,
       "CitizenAccount_listPayments", "paymentDetails");

   participantMessage.addLink(participantMessageLink);
```

Before composing the message, the dynamic listener must check to ensure that the message type in question is currently enabled. The `curam.participantmessages.configuration.impl.ParticipantMessageConfiguration` record for that message type should be read, and the `isEnabled` method used to determine if this message type is enabled. If not, no further processing should occur.

* It is recommended to separate the code that listens for the event and the code that composes a specific message, in order to adhere to the philosophy of "doing one thing and doing it well".

Implementing a persisted message

In order to have a persisted message displayed to the citizen, it must be written to the database via the `curam.citizenmessages.persistence.impl.ParticipantMessage` API. Message arguments are handled by persisting a `curam.advisor.impl.Parameter` record and associating it with the ParticipantMessage record, and links by the `curam.advisor.impl.Link` API. Parameter names should map to placeholders contained within the message text. Link names should relate to the names of links specified in the message text. Please refer to the Javadoc information of `curam.citizenmessages.persistence.impl.ParticipantMessage`, `curam.advisor.impl.Parameter` and `curam.advisor.impl.Link` for more.

An expiry date time must be specified for each ParticipantMessage. After this date time, the message will no longer be displayed.

Messages can be removed from the database. If a message needs to be replaced with a modified version, or removed for another reason, this can be done via the `curam.citizenmessages.persistence.impl.ParticipantMessage` API.

Each message has a related ID and type. This is used to track the record that the message is related to. For example, meeting messages will store the Activity ID and a type of "Meeting". Messages can be read by participant, related ID and type via the `ParticipantMessageDAO`.

Before persisting the message, the dynamic listener must check to ensure that the message type in question is currently enabled. The `curam.participantmessages.configuration.impl.ParticipantMessageConfiguration` record for that message type should be read, and the `isEnabled` method used to determine if this message type is enabled. If not, no further processing should occur.

Customizing specific message types: You can customize the default message types in various ways. See the `Cúram Universal Access Guide` for a description of the various message types.

Referral Message

This message type creates messages related to referrals. This is a dynamic message. When the citizen logs in, a message will be created for each referral that exists for the citizen in the system, provided that referral has a referral date of today or in the future, and provided that a related Service Offering has been specified for this referral. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageReferral.properties` contains the properties for the referral message text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered under the resource name `CitizenMessageReferral`. To change the message text of the message, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store.

Service Delivery Message

This message type creates messages related to service deliveries. This is a dynamic message. When the citizen logs in, a message will be created for each service delivery that exists for the citizen in the system, provided that service delivery has a status of 'In Progress' or 'Not Started'. The properties file `EJBServer\components\`

`CitizenWorkspace\data\initial\blob\prop\`
`CitizenMessageServiceDelivery.properties` contains the properties for the service delivery message text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered under the resource name `CitizenMessageServiceDelivery`. To change the message text of the message, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store.

Payment Messages: This message type creates messages based on the payments issued, canceled, an so on, for a citizen. These messages are persisted to the database. They replace each other, for example, if a payment is issued and then canceled, the payment issued message will be replaced with a payment canceled message. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageMyPayments.properties` contains the properties for financial message text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered under the resource name `CitizenMessageMyPayments`. To change the message text of financial messages, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store. The table below describes the messages created when various events related to payments occur in the system, and the property in `CitizenMessageMyPayments.properties` that relates to each message created.

Table 14. Payment messages and related properties

Payment event	Message Property
First payment issued on a case	Message.First.Payment
Latest payment issued	Message.Payment.Latest
Last payment issued	Message.Last.Payment
Payment canceled	Message.Cancelled.Payment
Payment reissued	Message.Reissue.Payment
Payment stopped (case suspended)	Message.Stopped.Payment
Payment / Case unsuspended	Message.Unsuspended.Payment

Customization of the Payment Messages Expiry Date

The number of days the payment for which the message will be displayed to the citizen can be configured using a system property. By default the property value is set to 10 days, however, this can be overridden from property administration.

Table 15. Payment message expiry property

Name	Description
curam.citizenaccount.payment.message.expiry.days	The number of days the payment message will be displayed to the participant.

Meeting Messages: This message type creates messages based on meetings that the citizen is invited to, provided that they are created via the `curam.meetings.sl.impl.Meeting` API. This API raises events that the meeting messages functionality consumes. There are other ways of creating Activity records without this API, but meetings created in these ways will not have related messages created as the events will not be raised. These messages are persisted to the database. They replace each other, for example, if a meeting is scheduled and then the location is changed, the initial invitation message will be replaced with one informing the citizen of the location change. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageMeetingMessages.properties` contains the properties for the meeting messages text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered under the resource name `CitizenMessageMeetingMessages`. To change the message text of meeting messages, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store. The table below describes the messages created when various events related to meetings occur in the system, and the properties in `CitizenMessageMeetingMessages.properties` that relates to each message created. Different versions of the message text are displayed depending on whether the meeting is an all day meeting, whether a location has been specified, and whether the meeting organizer has contact details registered in the system. Accordingly, the property values in this table are approximations that relate to a range of properties within the properties file. Please refer to the properties file for a full list of the message properties.

Table 16. Meeting messages

Meeting event	Message Properties
Meeting invitation	Non.Allday.Meeting.Invitation.*, Allday.Meeting.Invitation.*
Meeting update	Non.Allday.Meeting.Update.*, Allday.Meeting.Update.*
Meeting canceled	Allday.Meeting.Update.*, Allday.Meeting.Cancellation.*

Customization of the Meeting Messages Display Date

The number of days before the meeting start date that the message should be displayed to the citizen can be configured using a system property. By default the property value is set to 10 days, however, this can be overridden from property administration.

The meeting message expires (i.e. it is no longer displayed to the citizen) at the end of the meeting, i.e. the date time at which the meeting is scheduled to end.

Table 17. Meeting message display date property

Name	Description
curam.citizenaccount.meeting.message.effective.days	The number of days before the meeting start date that the message should be displayed to the citizen.

Customization of Activity types for which to create Meeting Messages

Meetings are stored on the Activity entity. There are different types of Activity, which are stored in the `CT_ActivityType` codetable. The list of activity types for which to create messages can be customized using the following property. The default code is 'AT2' which represents Meeting.

Table 18. Activity types for which to generate meeting messages

Name	Description
curam.citizenaccount.meeting. activity.types.to.generate.messages	A configuration setting to dictate the types of activities for which messages will be generated.

Application Acknowledgment Message: This message type creates a message when an application is submitted by a citizen. This message is persisted to the database. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageApplicationAcknowledgment.properties` contains the properties for the messages text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered under the resource name

`CitizenMessageApplicationAcknowledgment`. To change the message text of the message, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store.

Customization of Application Acknowledgment Message Expiry Date

The number of days the Application Acknowledgment message will be displayed to the citizen can be configured using a system property. By default the property value is set to 10 days, however, this can be overridden from property administration.

Table 19. Application acknowledgment message expiry property

Name	Description
curam.citizenaccount. intake.application.acknowledgement.message.expiry.days	The number of days the application acknowledgment message will be displayed to the participant.

Customizing existing pages

The Notices, My Payments, My Applications and My Activities pages that are shipped OOTB are customizable. The UIM pages can be replaced higher up the component order and changes made as required, as with standard UIM pages.

On the server side, the APIs that drive these pages are customizable and live in the `curam.citizenaccount.impl` package. These can be customized along with the structs that they return in order to return additional information. This is preferential to customizing the `curam.citizenaccount.facade.impl.CitizenAccount` facade, which is internal and should not be called.

The required security checks live in the `CitizenAccount` facade as opposed to in the APIs located in `curam.citizenaccount.impl`. Custom facades must implement the required checks.

The structs that are used to serve these pages with data are also customizable. The model files that contain these structs are located at `EJBServer\components\CitizenWorkspace\model\Packages\CitizenAccount`. Unlike the other model artifacts in UA, these are customizable in the standard way.

Please note, there are constraints on the use of UIM in Universal Access. These constraints are discussed in the section "Create a UIM page in the new component" on page 25.

Customizing the Notices Page

Complete the following the steps to customize the Notices page.

Listing of Communication records

curam.citizenaccount.impl.CitizenCommunicationsStrategy.listCitizenCommunications(ConcernRoleK has default implementation for listing the Citizen Communication records i.e., the notices belonging to the logged in user will be listed by default on the Notices page. This can be modified with other custom implementations. For e.g., one can retrieve the communications of all the household members of the logged in citizen through their own custom implementation.

Communication events

curam.core.events.CONCERNROLEACOMMUNICATION.INSERT_CONCERN_ROLE_COMM

curam.core.events.CONCERNROLEACOMMUNICATION.MODIFY_CONCERN_ROLE_COMM

These are the events that are raised post creation or post modification of a communication record. Custom implementations can listen to these events for any kind of post processing requirements.

Communication Processing Hooks:

The manner in which electronic notices are managed and supported in the Citizen Portal has a knock-on impact on the communication processing module. Whereas OOTB doesn't address or implement any of the impacts, the following OOTB hooks are available for the custom implementation to customize the communication processing module.

curam.core.hook.impl.PreCreateCommunicationHook - can be used in customized scenarios for any kind of pre creation processing for communication records.

curam.core.hook.impl.PreModifyCommunicationHook - can be used in customized scenarios for any kind of pre modify processing for communication records.

For e.g.; in situations where create or modify operation is not applicable, this hook points can be used to redirect the user with customized messages before the creation or modification of communication records using custom exception handling.

curam.core.hook.impl.CommunicationInvocationStrategyHook - can be used as a toggle the above hooks i.e., PreModifyCommunicationHook and PreCreateCommunicationHook should be invoked or not.

The following communication processing methods have been updated by the pre creation and pre modification hooks that are mentioned above to enable further customization.

- curam.core.facade.impl.Communication.modifyWordDocument(ModifyWordDocumentDetails)
- curam.core.facade.impl.Communication.modifyEmail(ModifyEmailCommDetails, ModifyEmailCommKey)
- curam.core.facade.impl.Communication.modifyRecordedCommunication1(ModifyRecordedComm
  ModifyRecordedCommDetails1)
- curam.core.facade.impl.Communication.modifyProForma1(ModifyProFormaCommDetails1)
- curam.core.facade.impl.Communication.createEmailCommunication(CreateEmailCommDetails)
- curam.core.facade.impl.Communication.createEmail(CreateEmailCommDetails)
- curam.core.facade.impl.Communication.createMSWordCommunication1(CreateMSWordCommun
- curam.core.facade.impl.Communication.createCaseMSWordCommunication1(CreateMSWordCom

- curam.core.facade.impl.Communication.createRecordedCommunication1(RecordedCommDetails1)
- curam.core.facade.impl.Communication.createProForma1(CreateProFormaCommDetails1)
- curam.core.facade.impl.Communication.createProFormaCommunication1(CreateProFormaCommDe

Availability of APIs:

This section lists out the available APIs for custom communication requirements.

Various APIs are now available for the usage of custom requirements such as

curam.citizenaccount.facade.impl.CitizenAccount.listCitizenCommunications()

The content for the Notices page is retrieved using the above API. This API in turn delegates the call to the strategy that is available for customization/extension.

curam.core.facade.impl.Communication.sendByEmailWithNoRecord(EmailCommunicationDetails)

This APIs can be used for any custom implementations that are having a need to send an alert to the user through email for notices posted on the citizen account.

curam.core.facade.impl.ConcernRole.readConcernRolePreferredCommunication(ConcernRoleKey)

curam.core.facade.impl.ConcernRole.modifyConcernRolePreferredCommunication(ConcernRoleK ConcernRolePreferredCommunication)

These provide the ability for any custom implementations to read or specify preferred communication mode.

For e.g., any custom implementation having a need to specify the preferred communication as electronic means then they can use the above modify API and implement custom UI option to call this method.

curam.citizenaccount.facade.impl.CitizenAccount.markCommunicationToSendByMail(ConcernRol

This API will record the request initiated by the user when 'Send by Post' link is clicked using the table 'CitizenCommSendByPostTracker'. This will provide the ability for any custom implementations to write any workflows/batch jobs to satisfy the requirement of sending the selected communication in their citizen account through the regular **post/mail** .

curam.citizenaccount.impl.CitizenCommunicationsStrategy.searchCitizenCommSendByPostTracke

This API will return all the records from 'CitizenCommSendByPostTracker' table which are marked to be sent by post for a concern role.

Enabling the Notices tab

The Notices tab on the Citizen Portal allows users to view their communications online. The Notices tab is shown by default in Cúram Universal Access. For solutions that use a custom Citizen Portal, such as Curam Income Support for Medical Assistance (Health Care Reform), you can manually enable the Notices tab.

About this task

When enabled, the Notices tab is displayed on the navigation bar in Universal Access. The Notices tab has a list of all types of communications either sent to the citizen by the agency or sent to the agency by the citizen.

Procedure

1. Edit the custom `CitizenAccount.nav` file and enter the following code:

   ```
   <nc:navigation-page id="mycommunications"
       page-id="CitizenAccount_communications"  title="leaf.title.communications"
       icon="CitizenAccount.communications.notices.leftnav.icon"
       description="CitizenAccount.desc"/>
   ```
2. Edit the custom `CitizenAccount.properties` file and enter any required properties.

Moving the LogOut button to improve usability

The LogOut button from the Welcome Person menu is now available in the Universal Access mega-menu for improved accessibility. For solutions that use a custom Citizen Portal, such as Curam Income Support for Medical Assistance (Health Care Reform), you can manually implement this improvement.

About this task

For Cúram Income Support for Medical Assistance (Health Care Reform), the LogOut button is in a drop-down menu next to the welcome text on the mega-menu bar. The text string 'LogOut' displayed on the user interface is specified by the value of the title attribute of the banner-menu element added to the associated CITWSAPP.properties file.

If you move the LogOut option, you can choose to retain or remove the previous LogOut option from the welcome person drop-down menu.

Procedure

1. Edit the custom `CITWSAPP.app` file for the new LogOut button and enter the following code:

   ```
   <ac:banner-menu type="logout" title="logout.title" />
   ```
2. Edit the custom `CITWSAPP.properties` file for the new LogOut button and enter any required properties.

My Payments Page Customization

Data for this page is retrieved using the `curam.citizenaccount.impl.CitizenPayments` API. The `listPayments` method is used to list the payments on the page. The in line instruction details page calls the `readPaymentInstructionByInstrument` method to retrieve the payment instruction details.

My Applications Page Customization

Data for this page is retrieved using the `curam.citizenaccount.impl.CitizenPayments` API. The `listPayments` method is used to list the payments on the page. The in line instruction details page calls the `readPaymentInstructionByInstrument` method to retrieve the payment instruction details. Please consider the required security checks when consuming this API in custom facades.

Contact Information Page Customization

This page displays case worker contact details for each Case related to the citizen, along with the citizen's contact information.

It is customizable in a number of ways, described in the table below.

Table 20. Contact Information Customization properties

Name	Description	Default
curam.citizenaccount. contactinformation.show.caseworker.details	Whether to display case worker contact information on this page.	true
curam.citizenaccount. contactinformation.show.casemember.cases	Whether to display case worker details where the citizen is a case member, as opposed to the primary participant.	true
curam.citizenaccount. contactinformation.show.businessphone	Whether to display the case workers' business phone number.	true
curam.citizenaccount. contactinformation.show.mobilephone	Whether to display the case workers' cell / mobile phone number.	true
curam.citizenaccount. contactinformation.show.faxnumber	Whether to display the case workers' fax number.	true
curam.citizenaccount. contactinformation.show.pagernumber	Whether to display the case workers' pager number.	true
curam.citizenaccount. contactinformation.show.emailaddress	Whether to display the case workers' email address.	true

Adding additional contact information.

A customer might want to display additional contact information, e.g. Twitter handle, for a case worker. To do this a customer should implement `curam.citizenaccount.impl.CitizenContactHelper` interface. This interface allows the addition of extra case worker contact details to the Contact Information screen in the Citizen Account. The `CaseWorkerContacts.name` attribute resolves to a property entry in `CitizenAccountContactInformation.properties` which must be defined. For example `CaseWorkerContacts.setName("twitterhandle");` would require an entry similar to `twitterhandle=Twitter` in `CitizenAccountContactInformation.properties`.

Customizing Appeal Requests

Complete the following steps to customize Appeal Requests in the Citizen Account to your requirements.

Displaying appeals request status from an external appeals system

You can create an implementation to enable the display of appeal request status from an external appeals system in the citizen account by using the provided API.

About this task

The curam.core.onlineappealrequest.impl.OnlineAppealRequestStatus interface takes an appeal request as an input and passes back a code table value. You can modify code table entries as required.

Procedure

1. Identify the appeal request ID from the caseworker application.
2. Use the appeal request ID to associate the appeal request status from the external system with the appeal request status in Universal Access.

3. Implement the curam.core.onlineappealrequest.impl.OnlineAppealRequestStatus interface to return the appropriate code table value based on the OnlineAppealRequest. For example, a custom implementation of this class might call a remote system and map the return value to an appropriate code table value.
4. Customize an appeal status message to display in the Citizen Account.

Customizing Life Events

A description of the high-level architecture of Life Events and how to perform the analysis and development tasks in building a Life Event. Use this information to understand Life Event and why they are useful, and how to develop Simple Life Events.

Many types of Life Events can be built entirely by analysts, some will require input from developers. This information will help analysts to understand how to perform the analysis for a new Life Event and how to determine whether input is needed from developers.

Introduction to Life Events

Life Events are intended to capture a holistic view of what is happening in a person's life. Life Events provide, not only raw information about a person's circumstances, income, and so on, but also context.

Consider the following scenario: James Smith loses his job after the company he is working with shuts down. James logs in to his Citizen Account and goes to the Life Event section. He chooses the **Lost my Job** Life Event. The system starts an intelligent evidence gathering (IEG) script to collect information about the Job Loss event. The script asks James a number of relevant questions about the circumstances of his Job Loss.

These questions are not necessarily relevant to any particular Social Assistance Program that James might be on. A Life Event Script is typically short and to the point. Some of the information in the script might be pre-filled with information that is already known about James Smith. For example, the name and address of his former employer are displayed in the script (this is known as pre-population of the IEG script). James confirms that indeed this is the employer that laid him off.

After the Life Event script is completed, a set of recommendations is displayed. These recommendations include:

- Services in the community that can provide him with help and support
- Government run Programs that might be relevant to James' situation, for example Unemployment Benefit

A couple of days after the Life Event is submitted, James logs in to his Citizen Account again. He sees a message on his home page. James is on a Benefit Case, and as a result of the changes in the Life Event the agency that administers this benefit needs to collect some more information about James' income. After completing another question script, James returns to the Life Event pages and reviews information about his previously submitted "Lost my Job" Life Event. He can see the information that he sent to the agency and also remind himself of the services that are recommended as a result.

From James' point of view he has:

- Told one or possibly several different agencies about his misfortune, he hasn't had to contact them separately
- He has been recommended services that are in his community and close to where he lives. He might not have been even aware that such services existed before
- He has been recommended to apply for appropriate government programs

From the point of view of interested Social Enterprises:

- They get context. They not only know that James applied for a program, they now know what prompted him to apply
- James has been triaged by the Citizen Account system, saving valuable resources
- James has been directed towards community / voluntary resources that can help him
- If James has existing cases that are being managed by using Cúram, then information from the Life Event can be fed automatically into these cases

How to Build a Life Event

Details of how to perform the analysis and development tasks in building a Life Event.

Analysis

You must undertake an analysis in order to design a Life Event for Universal Access. It is possible to build Life Events for case workers or indeed to use Life Event infrastructure to drive other processes like certification, but these topics are beyond the scope of this information. Java coding skills are not a prerequisite for developing all Life Events. Depending on requirements, many and in some cases all of the artifacts required can be developed by an Analyst. This topic will help Analysts to determine whether Java developers will be needed to complete the implementation of a Life Event.

Broadly speaking, Life Events for Citizens come in two flavors:

- Standard Life Events
- Round Tripping Life Events

Standard Life Events allow the Citizen to enter new information and then submit it to the agency. For example: Imagine, that Linda logs in to Universal Access and submits a "Having a Baby" life event. This is all new information, it doesn't really need relate to anything that has gone before. If it turns out that she has made a mistake in the information she submitted, say the name of the obstetrician, then she simply launches a new Life Event and re-enters all the new information again before submitting.

Round Tripping Life Events are more complex. The distinction between these Life Events and Standard Life Events is determined by whether the data that is pre-populated into the Life Event is allowed to be changed by the user. If the Citizen is expected to update pre-populated information, rather than just adding new information then the Life Event should be considered a Round Tripping Life Event. It is considerably harder to design scripts for this type of Life Event.

The primary artifacts that constitute a Simple Life Event are:

- An IEG script and its associated data store schema
- An IEG script to review answers in a previously submitted Life Event (optional)

- A Cúram Data Mapping Engine specification that describes how to map data from the IEG script into evidence on the client's cases

All of these artifacts can be configured using the Administrator's User Interface. For more information about configuring Simple Life Events using the Administrator's UI, see "Configuring Life Events" in the IBM Cúram Universal Access Configuration Guide.

The Life Events system can take information entered by the user and do either of two things with that information:

1. If the user is linked to the local Cúram case processing system, then the Life Events system can update related evidence in any cases they have.
2. If the user is linked to remote systems then the Life Events system can send updates to related remote systems via web services.

If the Life Event is a Round Tripping Life Event or it is required to update the person's evidence in Cúram then some development work will be needed. See the Life Events APIs needed to meet these requirements or indeed to supplement the standard Life Event behavior with additional custom functionality.

Customizing Advanced Life Events

Use this information to understand what distinguishes an Advanced Life Event from a Simple Life Event, and how to develop Advanced Life Events. This information describes the high-level architecture of Advanced Life Events and details how to perform the analysis and development tasks in building an Advanced Life Event. It also describes the Advanced Life Events Java API.

Advanced Life Events and when to use them

Advanced Life Events enable fully automated round-tripping of data. This means that client evidence is read into the datastore for an IEG script. It is then updated by the client. When the Life Event is submitted, the original client evidence that was read into the IEG script is updated. Advanced Life Events are only required when this level of automated round tripping of data is required. Under all other circumstances Simple Life Events are the recommended approach. Project Architects should consider carefully whether round tripping is required or whether the data entered by a client can be treated as new evidence to be integrated into the client's cases.

Advanced Life Events cannot be configured through the Administration user interface, they must be created by developers.

How to Build a Life Event

Analysis

The distinction between these Round Tripping Life Events and Standard Life Events is determined by whether the data that is pre-populated into the Life Event is allowed to be changed by the user. If the Citizen is expected to update pre-populated information, rather than just adding new information then the Life Event should be considered a Round Tripping Life Event. It is considerably harder to develop this type of Life Event. The Advanced Life Events subsystem is designed to cater for Round Tripping Life Events. The following information describes how to develop an Advanced Life Event that supports Round Tripping of the client's information.

The primary artifacts that constitute an Advanced Life Event are:

- An IEG script and its associated data store schema
- An IEG script to review answers in a previously submitted Life Event (optional)
- An Recommendations Ruleset, that produces the set of recommendations based on the information entered in the IEG script (optional)

The Life Events system can take information entered by the user and do either of two things with that information:

1. If the user is linked to the local Cúram case processing system, then the Life Events system can update related evidence in any cases they have.
2. If the user is linked to remote systems then the Life Events system can send updates to related remote systems via web services.

The Life Events system can be configured to seek the user's permission before sending Life Event information to any remote systems.

A standard Life Event that is configured only to send information to remote systems can be configured through the administration application. See Universal Access Configuration Guide for details.

If the Life Event is a Round Tripping Life Event or it is required to update evidence in the local case processing system then some development work will be needed to configure the Life Event. Round Tripping Life Events must be pre-populated. Currently pre-population of Life Events is only supported for users linked to the local Cúram case processing system via a concern role. To read information from cases and update those cases, the Life Events system relies on a subsystem called the Citizen Data Hub.

The remainder of this topic outlines the work needed to configure the Citizen Data Hub.

The Life Event Broker uses the Data Hub to get the data it needs to populate the Life Event, so the developer must configure the Data Hub to extract this data. The Life Event Broker also sends the updated data back through the Data Hub. The Data Hub must be configured to tell it what to do with this updated data.

These are some of the artifacts used to configure the Citizen Data Hub for reading information:

- Transform - Translates data from the Holding Case into Data Store XML
- Filter Evidence Links - When reading Citizen Data, these links filter out only the evidence entities of interest when reading from the Holding Case
- View Processors - Java classes for extracting non-evidence data into the Data Store XML

These are some of the artifacts that are used to configure the Citizen Data Hub for updating information:

- Transforms - Convert a Data Store XML Difference Description back into Holding Case Evidence
- Update Processors - Perform other update tasks or update non-evidence data relating to the Citizen

Considerations for Life Events Analysis:

Here are some of the considerations that affect the complexity of developing a particular Life Event that must read from, or write to, an evidence or participant-related data store in Cúram. These considerations inform any analysis of Life Events development and any resulting estimates.

1. Is the Life Event a Standard Life Event or a Round Tripping Life Event
2. What information needs to be pre-populated into the IEG script?
3. What Evidence data is read by the Life Event?
4. What Evidence data is updated by the Life Event?
5. What non-Evidence data is read/updated by the Life Event
6. How many Programs/Case Types are affected by the Life Event
7. If a Life Event shares to multiple Cases, will those case types also share evidence with each other using Evidence Broker?
8. Does a Life Event have associated Recommendations? If so, do they relate to Community Services, Government Programs or both?

Of these items that deal with Non-Evidence Entities presents the greatest challenge. Any Life Event that updates non-evidence entities require developers with Java skills.

Building The Components of a Life Event

A description of how to build the component parts of a Life Event that uses the Citizen Data Hub. This information does not require any knowledge of Java.

- How to write Life Event IEG Scripts, including Review Scripts
- How to write Life Events Recommendations Rule Sets
- How to pre-populate a Life Event Script using the Citizen Data Hub
- How to process Life Event Updates using the Citizen Data Hub
- How to put all the components together

Overview: An outline of how to build the component parts of a Life Event that uses the Citizen Data Hub. This information does not require any knowledge of Java.

- How to write Life Event IEG Scripts, including Review Scripts
- How to write Life Events Recommendations Rule Sets
- How to pre-populate a Life Event Script using the Citizen Data Hub
- How to process Life Event Updates using the Citizen Data Hub
- How to put all the components together

Writing Life Event IEG Scripts:

Writing a Life Event intelligent evidence gathering (IEG) script is similar to writing any other IEG scripts. However, some special considerations exist for Life Event scripts. Mostly, these considerations depend on whether the Life Event is a Round Tripping Life Event or a Standard Life Event.

For a Round Tripping Life Event, citizen data is read into the data store that is used by the IEG script. This data can be modified by citizens as they progress from page to page in the Life Event script. For example, a citizen can modify a piece of Income data that is read into the Life Event script before it is submitted. The Life Event Broker ensures that when the citizen changes the Income data the changes are detected and propagated correctly back to the Income entity from which the data was read originally. The Life Event Broker needs a way to track data from its origin in the Income entity, through the Life Event Script, and back to the same Income entity. To facilitate this process, the IEG script designer needs to place a marker into the data store schema.

The following code block is an example of the definition of an Income data store:

```
<xsd:element name="Income">
     <xsd:complexType>
        <xsd:attribute name="incomeType" type="INCOME_TYPE"
           default=""/>
       <xsd:attribute name="cgissIncomeType"
            type="CGISS_INCOME_TYPE"/>
       <xsd:attribute name="incomeFrequency"
            type="INCOME_FREQUENCY" default=""/>
      <xsd:attribute name="incomeAmount" type="IEG_MONEY"
           default="0"/>
       <xsd:attribute name="localID" type="IEG_STRING"/>
     <xsd:complexType>
  </xsd:element>
```

The attribute `localID` is used by the Cúram Life Event Broker to track the unique identity of the entity from which the Income Data was drawn. When this entity is changed by the user and submitted, the Life Event Broker can use the value of `localID` to locate the correct entity to update as a result of the changes in the Life Event. Other special markers exist that can be placed in the schema to aid with providing automatic updates to Cúram evidence entities.

When a user designs a script for a Round Tripping Life Event, the designer needs to account for the effects that pre-population of data can have on the flow of the script. One example of this situation is conditional clusters. Life Event Scripts need to avoid conditional clusters that are associated with pre-populated data. These clusters are common in Intake scripts but do not work well when the data store was pre-populated. For example, for a Life Event involving the loss of a job, a boolean flag on the `Person` entity, `hasJob` is used to indicate that person has a job. The IEG script presents the user with a question: `Does anyone in your household have a job?`. This question is used to drive the display of a conditional cluster that identifies which household members who have jobs.

However, if the data in the data store is repopulated, it is likely one or more `Person` entities with `hasJob` already be set to `true`. In the current implementation of IEG, it is not possible to get the `Does anyone in your household have a job?` control question to default to `true` even when `hasJob` is `true` for one or more household members. For this reason, the rule needs to be to avoid control questions for conditional clusters such as when the fields they control are pre-populated.

Writing Life Event Review Scripts

Users who previously submitted a Life Event can return to review the answers they gave. IEG scripts are an ideal way to present this kind of information in a page-by-page, easily readable format. Still, a script that is suitable for data collection necessarily is not suited for use in the review of previously submitted data. For one thing, the fields should not be editable in a review script. IEG provides a Summary Page feature that can be used for rendering summaries of data that was entered previously. Summary pages are recommended as a good way of writing Life Event Review scripts. If a review script is not supplied, the question script is started in read-only mode when a user elects to review their Life Event.

Writing Life Event Recommendations Rule Sets: After submitting a Life Event the user is presented with a Screen showing Community Services in their area that are deemed suitable based on the Life Event they have just submitted. The same screen can also list recommended Government Programs for which to Self Screen or perform Intake. Life Event Recommendations Rule Sets must extend the `TriageInterface` rule set and extend `AbstractTriageResult`. As follows:

```
<RuleSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
  "../../../../CREOLEInfrastructure/xsd/curam/
    creole/xsd/RuleSet.xsd"
  name="LifeEventRecommendationsRuleSet">
  <Class name="LifeEventRecommendation"
    extends="AbstractTriageResult"
    extendsRuleSet="TriageInterfaceRuleSet">
    ...
  </Class>
  ...
</RuleSet>
```

For the most part, writing a Life Event Recommendations rule set resembles writing a Triage Rule Set, see "Customizing Triage". Where Rules for Life Event Recommendations differ is that they can make decisions based on whether a given Data Store entity was changed by the user executing the Life Event Script and, if it was changed, what was the nature of the change. For example, the Rule Set could make one recommendation based on the addition of a new Income entity or a different one based on a change to an existing Income Entity. The example below shows how to add rule attributes in support of Life Event Recommendations to a `Person` class.

```
<Class name="Person" xsi:noNamespaceSchemaLocation=
  "http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Attribute name="curamDataStoreUniqueID">
    <type>
      <javaclass name="Long"/>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>

  <Attribute name="curamHasChanged">
    <type>
      <javaclass name="Boolean"/>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>

  <Attribute name="curamChangeType">
    <type>
      <javaclass name="String"/>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>
</Class>
```

Following the submission of a Life Event, the Life Event Broker initializes a Rule Session and creates Rule Objects corresponding to the Data Store Entities for the Life Event. It then modifies these Rule Objects based on the Difference Command that corresponds to that Data Store entity. Taking the example of the Person Rule

Class described above: If the Person entity in the data store was changed as a result of the execution of the Life Event IEG script then the `curamHasChanged` attribute will return `true`. The `curamChangeType` will return the type of change that was made:

- `DCMDT10001` - The entity was added by the Life Event IEG Script
- `DCMDT10002` - The entity was changed by the Life Event IEG Script
- `DCMDT10003` - The entity was removed by the Life Event IEG Script

Pre-Populating a Life Event: A description of the artifacts that need to be developed in order to pre-populate a Life Event script:

- How the Data Hub Works for reading data
- How to author Read Transforms
- How to use Pre-Packaged View Processors

How the Data Hub Works for Reading

The Data Hub is a means of collecting data about Citizens from many different locations and returning it as an XML document in a datastore. The Data Hub can be used to hide the complexities of where data comes from and how it is represented in it original locations. For example, to drive a "Lost my Job" Life Event it might be necessary to gather information about a person's Income, Address and Employment. These three pieces of information might be represented differently on the underlying system, indeed they might live on three or more different systems. The caller doesn't need to know this. The Citizen Data Hub allows its clients to get these pieces of information in one single operation. Operations of this type are named uniquely, each is called a "Data Hub Context". To animate the "Lost my Job" example we define a Data Hub Read Context called "CitizenLostJob" that allows the collection of Income, Address and Employment information in a single query.

One of the sources that the Data Hub can draw on is Evidence on Cases. In particular, Evidence on the Citizen's Holding Case. The Holding Case can use the Evidence Broker to gather data from many disparate Integrated Cases or even from other Systems via Web Services. The Holding Case is a little different from other Cases. There is only ever one per Citizen on a given Cúram system. The Holding Case has an interface that allows all of the Evidence it contains to be extracted in XML format. This XML format is optimized for the description of Evidence in particular. Because it is optimized for the description of Evidence, it isn't necessarily in a format suitable for insertion into a data store. Fortunately it is relatively easy to translate data in one XML format into another format that contains the same information. This can be done using a language called XSLT For more information on XSLT please refer to, http://www.w3.org/TR/xslt.

Authoring Read Transforms

You can write XSLT Transforms for use in the Data Hub. To write Citizen Data Hub Transforms it is necessary to understand, the structure of the Holding Evidence XML that is the source data and the Data Store schema that is the target. The "CitizenLostJob" Life Event is significantly complex so, for the purposes of an introductory example, this section describes a simple fictitious Life Event for Citizens who have bought a new car. This Life Event is associated with the Data Hub Context "CitizenBoughtCar". This would not be considered a "Life Event" in the real world but it nevertheless provides an example of some of the principles of building a Round Tripping Life Event. For the purposes of this example consider this fragment of Holding Evidence XML that is used to describe a Vehicle:

```
<?xml version="1.0" encoding="UTF-8"?>
  <client-data
  xmlns="http://www.curamsoftware.com/schemas/ClientEvidence">
    <client localID="101" isPrimaryParticipant="true">
      <evidence>
        <entity localID="-416020015578349568" type="ET10081">
          <attribute name="vehicleMake">VM2</attribute>
          <attribute name="versionNo">2</attribute>
          <attribute name="startDate">20110301</attribute>
          <attribute name="usageCode">VU1</attribute>
          <attribute name="amountOwed">3,200.00</attribute>
          <attribute name="numberOfDoors">0</attribute>
          <attribute name="evidenceID">
            -5315936410157449216
          </attribute>
          <attribute name="monthlyPayment">0.00</attribute>
          <attribute name="vehicleModel">159</attribute>
          <attribute name="year">2008</attribute>
          <attribute name="equityValue">0.00</attribute>
          <attribute name="endDate">10101</attribute>
          <attribute name="fairMarketValue">17,000.00</attribute>
          <attribute name="curamEffectiveDate">20110301
           </attribute>
        </entity>
      </evidence>
    </client>
  </client-data>
```

Figure 5. Holding Evidence XML Example

The `client` element represents data belonging to the participant with concern role id 101. In Cúram demo data this is James Smith. The client contains a single evidence entity of type `ET10081`. In the Cúram Common Evidence layer, `ET10081` is the Evidence Type identifier for Vehicle Evidence. The `localID` attribute plus the evidence type uniquely identifies the underlying evidence object for the Vehicle. This data has to be mapped to data store XML so that it can be used to populate an IEG Script. Consider how the above data is to be represented in data store XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<Application>
    <Person localID="101" isPrimaryParticipant="true"
      hasVehicle="true">
        <Resource resourcePageCategory="RPC4001"
          localID="-416020015578349568" vehicleMake="VM2"
          versionNo="2" amountOwed="3,200.00" vehicleModel="159"
          year="2008" fairMarketValue="17,000.00"
          curamEffectiveDate="20110301">
            <Descriptor/>
        </Resource>
    </Person>
</Application>
```

Figure 6. Data Store XML Sample

This XML data must conform to the schema used to build the IEG script. Notice that the XML above conforms to a schema that is a superset of the `CitizenPortal.xsd` schema. It is recommended that the `CitizenPortal.xsd` schema be used as a starting point for the schemas used in Customer Life Events. To these schemas need to be added the "marker" attributes needed for Life Events. These marker attributes include the use of `localID` as discussed previously. Datastore schemata for entities can also include the following special markers that are specialized for representing Evidence in the Holding Case: The following XSLT fragment shows how to transform Vehicle Holding Evidence into a corresponding Data Store Entity:

- curamEffectiveDate - This maps to the effective date of a piece of Cúram Evidence

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:x="http://www.curamsoftware.com/
     schemas/DifferenceCommand"
    xmlns:fn="http://www.w3.org/2006/xpath-functions"
      version="2.0">
    <xsl:output indent="yes"/>

    <xsl:strip-space elements="*"/>

    <xsl:template match="update">
        <xsl:for-each select="./diff[@entityType='Application']">
            <xsl:element name="client-data">
                <xsl:apply-templates/>
            </xsl:element>
        </xsl:for-each>
    </xsl:template>

    <xsl:template match="diff[@entityType='Person']">
        <xsl:element name="client">
            <xsl:attribute name="localID">
                <xsl:value-of select="./@identifier"/>
            </xsl:attribute>
            <xsl:element name="evidence">
                <xsl:apply-templates/>
            </xsl:element>
        </xsl:element>
    </xsl:template>

    <xsl:template match="diff[@entityType='Resource']">
        <xsl:element name="entity">

            <xsl:attribute name="type">ET10081</xsl:attribute>
            <xsl:attribute name="action">
                <xsl:value-of select="./@diffType"/>
            </xsl:attribute>
            <xsl:attribute name="localID">
                <xsl:value-of select="./@identifier"/>
            </xsl:attribute>
            <xsl:for-each select="./attribute">
                <xsl:copy-of select="."/>
            </xsl:for-each>


        </xsl:element>
    </xsl:template>


    <xsl:template match="*">
        <!-- do nothing -->
    </xsl:template>
</xsl:stylesheet>
```

Figure 7. XSLT Transform for Vehicle Resource Information

The Life Event author who adds this transform to their Life Event can turn Vehicle Evidence recorded on any Integrated Case into a Data Store format that can be displayed in an IEG script with all the information pre-populated from the Evidence Record.

Defining Filters for Evidence

When the Holding Case is called upon to return an XML representation of its evidence, by default it will return all evidence for the citizen concerned. This could be a very large query that returns much more information than is required. The purpose of a Filter Evidence Link is to define, for each Data Hub Context, which Evidence Types are of interest. A Filter Evidence Link can be defined by adding entries to a Filter Evidence Link dmx file. The example below shows a Filter Evidence Link dmx file that defines the information that should be returned for the "CitizenBoughtCar" Life Event:

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="FILTEREVIDENCELINK">
    <column name="FILTEREVLINKID" type="id" />
    <column name="FILTERNAME" type="text" />
    <column name="EVIDENCETYPECODE" type="text" />
   <row>
        <attribute name="FILTEREVLINKID">
            <value>175</value>
        </attribute>
        <attribute name="FILTERNAME">
            <value>CitizenBoughtCar</value>
        </attribute>
        <attribute name="EVIDENCETYPECODE">
            <value>ET10081</value>
        </attribute>
    </row>
</table>
```

Using Pre-Packaged View Processors

Up to this point has focused on how Transforms can be used turn Evidence data into Data store XML for use in a Life Event Script. However there are other important pieces of information that are not represented as Evidence. In general the Life Event author must develop custom Java code in order to populate any information that is not represented as evidence. Using Java it is possible to develop *View Processors* which can be used to extract non-evidence data and translate this data into data store xml. By associating these View Processors with the right Data Hub Context, they can add their information into the data store in addition to the data put there by the transforms. The Life Events Broker ships with some pre-packaged View Processors that are capable of inserting certain frequently used non Evidence Data.

- Household View Processor
- The Person Address View Processor

The Household View Processor will find all Persons related to the currently Logged in user and pull them into the data store along with information on how they are related to the logged in Citizen. This information is based on the CEF `ConcernRoleRelationship` entity.

The Person Address View Processor populates the most important details of the logged in Citizen, such as name and Social Security Number. It also pulls in the Residential and Mailing addresses of the logged in Citizen. Both the Household View processor and the Person Address View Processor can be used together in the same Life Event Context but the Person Address View Processor should be run after the Household View Processor. The excerpt below shows how to configure these two View Processors to execute for the "CitizenBoughtCar" Life Event.

```
<?xml version="1.0" encoding="UTF-8"?>
  <table name="VIEWPROCESSOR">
    <column name="VIEWPROCESSORID" type="id"/>
    <column name="LOGICALNAME" type="text" />
    <column name="CONTEXT" type="text" />
    <column name="VIEWPROCESSORFACTORY" type="text" />
    <column name="RECORDSTATUS" type="text"/>
    <column name="VERSIONNO" type="number"/>
    <row>
      <attribute name="VIEWPROCESSORID">
        <value>4</value>
      </attribute>
      <attribute name="LOGICALNAME">
        <value>CitizenLostJob0</value>
      </attribute>
      <attribute name="CONTEXT">
        <value>CitizenBoughtCar</value>
      </attribute>
      <attribute name="VIEWPROCESSORFACTORY">
        <value>
        curam.citizen.datahub.internal.impl.
        +HouseholdCustomViewProcessorFactory
        </value>
      </attribute>
      <attribute name="RECORDSTATUS">
        <value>RST1</value>
      </attribute>
      <attribute name="VERSIONNO">
        <value>1</value>
      </attribute>
    </row>
    <row>
       <attribute name="VIEWPROCESSORID">
         <value>5</value>
       </attribute>
       <attribute name="LOGICALNAME">
         <value>CitizenLostJob1</value>
       </attribute>
       <attribute name="CONTEXT">
         <value>CitizenBoughtCar</value>
        </attribute>
        <attribute name="VIEWPROCESSORFACTORY">
           <value>
           curam.citizen.datahub.internal.impl.
           +CustomPersonAddressViewProcessorFactory
           </value>
        </attribute>
        <attribute name="RECORDSTATUS">
          <value>RST1</value>
        </attribute>
        <attribute name="VERSIONNO">
          <value>1</value>
        </attribute>
      </row>
   </table>
```

Note the use of the `CONTEXT` field. This links the `ViewProcessor` to the "CitizenBoughtCar" Life Event Context. This ensures that this `ViewProcessor` is called whenever the "CitizenBoughtCar" Data Hub Context is called. Notice also the use of a `logicalName` which uniquely distinguishes each View Processor. View Processors for a given Data Hub Context are executed in lexical order, so a View Processor name with a `logicalName` of "AAA" for the DataHubContext "CitizenBoughtCar" will be executed before one with a `logicalName` of "AAB".

Driving Updates from Life Events: A description of the artifacts that need to be developed in order to process the data submitted from a Life Event script. This information describes:

- How the Data Hub Works for updating data
- How to author Update Transforms
- How to create new Case Participants from Update Transforms
- How to configure Evidence Brokering for the Holding Case

How the Data Hub Works for Updating

Just as the Citizen Data Hub has a notion of Data Hub Context for reading so also does it have Data Hub Contexts for updating. Life Events will typically use the same Data Hub Context name for the read and updates associated with the same Life Event, so the "CitizenBoughtCar" context describes, not only, a set of artifacts for pre-populating a "CitizenBoughtCar" Life Event script but also a set of artifacts for handling updates to the Citizen's data when the "CitizenBoughtCar" Life Event script is complete.

An update operation for a given Citizen Data Hub Context can lead to many different individual entities being updated in a single transaction. The artifacts, provided to a Data Hub following a script submission are:

- A Data Store root entity
- A Difference Command
- A Data Hub Context Name

The Data Store root entity is the root of the data store that has been updated via the Life Events IEG script. The Difference Command is an entity that describes how this data store is different to the one that was passed to the IEG script before it was launched. In other words it describes how the user has changed the data as a result of executing the Life Event Script. These differences are broken down into three basic types:

- Creations - The user has created a data store entity as a result of running the IEG script
- Updates - The user has updated an entity as a result of running the IEG script
- Removals - The user has removed an entity as a result of running the IEG script

Of these three, Creations and Updates are the most common. Allowing users to remove items in Life Events scripts should generally be considered bad practice. Standard Life Events tend to be characterized by a number of Creations whereas Round Tripping Life Events tend to be a mixture of Creations and Updates. The Difference Command is generated automatically by the Life Event Broker after a Life Event is submitted.

To turn a Data Hub Update Operation into automatic updates to evidence entities on the Holding Case we need to specify a Data Hub Update Transform. In cases where there is a requirement to update non-evidence entities, an Update Processor must be developed. These Update Processors involve Java code development.

Writing Transforms for Updating

Update Transforms, like Read Transforms are specified using a simple XSLT syntax. In order to write update Transforms, the author must understand both the input XML, and the output Evidence XML format. The following examples are built around a "CitizenHavingABaby" Life Event. This Life Event allows the user to report that they are due to have a baby. They can enter a number of unborn children to indicate, for example, that they are expecting twins. The user can also enter a due date and they can nominate a father for the unborn child. The father can be an existing case participant or someone else entirely. In the latter case they must enter name, address, Social Security Number etc. This Life Event is not a "Round Tripping" Life Event, it is concerned with the creation of new Evidence rather than the update of existing Evidence. The input to an Update Transform is an XML-based description of the Data Store Difference Command. A sample difference command XML for the "CitizenHavingABaby" is depicted below:

```
<update>
  <diff diffType="NONE" entityType="Application">
    <diff diffType="NONE" entityType="Person" identifier="102">
      <diff diffType="CREATE" entityType="Pregnancy">
        <attribute name="numChildren">1</attribute>
        <attribute name="dueDate">20110528</attribute>
        <attribute name="curamDataStoreUniqueID">385</attribute>
      </diff>
    </diff>
    <diff diffType="UPDATE" entityType="Person" identifier="101">
      <attribute name="isFatherToUnbornChild">true</attribute>
      <attribute name="curamDataStoreUniqueID">399</attribute>
    </diff>
  </diff>
</update>
```

The difference command XML corresponds node-for-node with the data store XML. Each *diff* node describes how the corresponding data store entity has been modified by the execution of the IEG script. The `curamDataStoreUniqueID` attribute identifies which data store entity has changed. The `diffType` attribute identifies the nature of the change, for example `CREATE`, `UPDATE`, `NONE` or `REMOVE`. Attributes that are listed are those that have changed or been added to each data store entity. In the above example, the user has registered a pregnancy to Linda Smith (concern role ID 102) with one unborn child, due on May 28 th 2011. The father is listed as being James Smith (concern role ID 101). For more information on difference command XML please see the schema in Difference Command XML Schema section. There are a couple of additional attributes and elements used when updating XML that are illustrated below:

```
<?xml version="1.0" encoding="UTF-8"?>
  <client-data>
    <client localID="102">
      <evidence>
        <entity type="ET10074" action="CREATE" localID="">
          <attribute name="numChildren">1</attribute>
          <attribute name="dueDate">20110528</attribute>
          <entity-data entity-data-type="role">
            <attribute type="LG"/>
            <attribute roleParticipantID="102"/>
            <attribute
              entityRoleIDFieldName="caseParticipantRoleID"/>
          </entity-data>
          <entity-data entity-data-type="role">
            <attribute type="FAT"/>
              <attribute roleParticipantID="101"/>
              <attribute participantType="RL7"/>
              <attribute
                entityRoleIDFieldName="fahCaseParticipantRoleID"/>
            </entity-data>
          <entity type="ET10125" action="CREATE">
            <attribute name="comments"> Unborn child 1</attribute>
            <entity-data entity-data-type="role">
              <attribute type="UNB"/>
              <attribute roleParticipantID="102"/>
              <attribute
                entityRoleIDFieldName="caseParticipantRoleID"/>
            </entity-data>
          </entity>
        </entity>
      </evidence>
    </client>
  </client-data>
```

Figure 8. Evidence XML with Updates

Note the use of the `action` attribute which describes the action to be taken to the underlying evidence, for example, to create the evidence or to update existing evidence. The next section will discuss the meaning of the `<entity-data>` element. An example of the XSLT used to transform the above difference XML into the above Evidence XML is depicted below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This script plucks out and copies all resource-related -->
<!-- entities from output built by the XMLApplicationBuilder -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:x="http://www.curamsoftware.com/
      schemas/DifferenceCommand"
    xmlns:fn="http://www.w3.org/2006/xpath-functions"
    version="2.0">
    <xsl:output indent="yes"/>
    <xsl:strip-space elements="*"/>
    <xsl:template match="update">
        <xsl:for-each select="./diff[@entityType='Application']">
            <xsl:element name="client-data">
                <xsl:apply-templates/>
            </xsl:element>
        </xsl:for-each>
    </xsl:template>
    <xsl:template match="diff[@entityType='Person']">
        <xsl:element name="client">
            <xsl:attribute name="localID">
                <xsl:value-of select="./@identifier"/>
            </xsl:attribute>
            <xsl:element name="evidence">
                <xsl:apply-templates/>
            </xsl:element>
        </xsl:element>
    </xsl:template>
    <xsl:template match="diff[@entityType='Pregnancy']">
        <xsl:element name="entity">
            <xsl:attribute name="type">ET10074</xsl:attribute>
            <xsl:attribute name="action">
                <xsl:value-of select="./@diffType"/>
            </xsl:attribute>
            <xsl:attribute name="localID">
                <xsl:value-of select="./@identifier"/>
            </xsl:attribute>
            <xsl:for-each select="./attribute">
                <xsl:copy-of select="."/>
            </xsl:for-each>
            <xsl:element name="entity-data">
                <xsl:attribute name="entity-data-type">
                  role
                </xsl:attribute>
                <xsl:element name="attribute">
                    <xsl:attribute name="type">LG</xsl:attribute>
                </xsl:element>
                <xsl:element name="attribute">
                    <xsl:attribute name="roleParticipantID">
                        <xsl:value-of select="../@identifier"/>
                    </xsl:attribute>
                </xsl:element>
                <xsl:element name="attribute">
                    <xsl:attribute name="entityRoleIDFieldName">
                      caseParticipantRoleID
                    </xsl:attribute>
                </xsl:element>
            </xsl:element>
            <xsl:element name="entity-data">
                <xsl:attribute name="entity-data-type">
                  role
                </xsl:attribute>
                <xsl:element name="attribute">
                    <xsl:attribute name="type">FAT</xsl:attribute>
                </xsl:element>
                <xsl:for-each select=
                "../../diff[@entityType='Person']/attribute[
                  @name='isFatherToUnbornChild'
                  and ./text()='true']">
                    <!-- Copy the participant id if a family -->
                    <!-- member is the father -->
                    <xsl:element name="attribute">
                      <xsl:attribute name="roleParticipantID">
                          <xsl:value-of select="
                            ../@identifier"/>
                        </xsl:attribute>
                    </xsl:element>
                </xsl:for-each>
                <!-- Copy details of absent parent -->
                <xsl:call-template name="absentFather"/>
                <xsl:element name="attribute">
                    <xsl:attribute name="entityRoleIDFieldName">
                      fahCaseParticipantRoleID
                    </xsl:attribute>
                </xsl:element>
            </xsl:element>
            <xsl:variable name="numBabies">
                <xsl:value-of select="attribute[
                  @name='numChildren'
                  ]/text()"/>
            </xsl:variable>
            <xsl:call-template name="unbornChildren">
                <xsl:with-param name="count" select="$numBabies"/>
            </xsl:call-template>
        </xsl:element>
    </xsl:template>

    <xsl:template name="unbornChildren">
        <xsl:param name="count" select="1"/>
        <xsl:if test="$count > 0">
            <xsl:element name="entity">
                <xsl:attribute name="type">ET10125</xsl:attribute>
                <xsl:attribute name="action">
                    <xsl:value-of select="./@diffType"/>
                </xsl:attribute>
                <xsl:element name="attribute">
                    <xsl:attribute name="name">
                      comments
                    </xsl:attribute>
                    Unborn child <xsl:value-of select="$count"/>
                </xsl:element>
                <xsl:element name="entity-data">
                    <xsl:attribute name="entity-data-type">
                      role
                    </xsl:attribute>
                    <xsl:element name="attribute">
                        <xsl:attribute name="type">
                          UNB
                        </xsl:attribute>
                    </xsl:element>
                    <xsl:element name="attribute">
                        <xsl:attribute name=
                           "roleParticipantID">
                            <xsl:value-of select="
                              ../@identifier"/>
                        </xsl:attribute>
                    </xsl:element>
                    <xsl:element name="attribute">
                        <xsl:attribute name=
                          "entityRoleIDFieldName">
                          caseParticipantRoleID
                        </xsl:attribute>
                    </xsl:element>
                </xsl:element>
            </xsl:element>
            <xsl:call-template name="unbornChildren">
                <xsl:with-param name="count" select="$count - 1"/>
            </xsl:call-template>
        </xsl:if>
    </xsl:template>

    <xsl:template name="absentFather">
        <xsl:element name="attribute">
            <xsl:attribute name="participantType">
                <xsl:text>RL7</xsl:text>
            </xsl:attribute>
        </xsl:element>

        <xsl:if  test="attribute[@name='fahFirstName']">
            <xsl:element name="attribute">
                <xsl:attribute name="firstName">
                    <xsl:value-of select="attribute[
                      @name='fahFirstName'
                    ]/text()"/>
                </xsl:attribute>
            </xsl:element>
        </xsl:if>

        <!-- etc. map other personal details such as -->
        <!-- SSN, date of birth -->
```

```
        <xsl:if  test="diff[@entityType='ResidentialAddress']">
            <xsl:if  test="diff[
              @entityType='ResidentialAddress']/attribute[
              @name='street1']">
                <xsl:element name="attribute">
                    <xsl:attribute name="street1">
                        <xsl:value-of select=
                        "diff[
                        @entityType='ResidentialAddress']
                          /attribute[
                        @name='street1']/text()"/>
                    </xsl:attribute>
                </xsl:element>
            </xsl:if>
            <!-- etc. map other parts of residential address -->
        </xsl:if>
    </xsl:template>

    <xsl:template match="*">
        <!-- do nothing -->
    </xsl:template>
</xsl:stylesheet>
```

Writing Transforms that create new case participants

Readers who are familiar with Evidence will know that Evidence Entities frequently refer to third parties. For example, Pregnancy evidence refers to the father via a Case Participant Role. The associated father can be a Person or a Prospect Person. Other evidence types such as `Student` may refer to a School which is entered as a Representative Case Participant Role.

The Evidence XML schema provides a generic element called `<entity-data>` which can be used to provide special handling instructions to the Citizen Data Hub. The type of handling depends on the `<entity-data-type>` specified. Cúram provides a special processor for the entity-data-type `role`. This role entity data processor can be used to create new Case Participant Roles or reference existing Case Participant Roles for existing Case Participants. Referring to the Evidence XML output in listed in the previous section the attribute denoted by `type` is used to denote the Case Participant Role Type e.g. FAT for Father or UNB for Unborn Child. The value provided here should be a codetable value from the `CaseParticipantRoleType` code table. The `roleParticipantID` denotes the ConcernRoleID of an existing participant on the system. If this is supplied then the system will not attempt to create a new Case Participant, rather it will reuse a case participant with this id. The `entityRoleIDFieldName` is the field name in the corresponding Evidence Entity. In the case of the `Pregnancy` evidence entity for example, the name of this field is `fahCaseParticipantRoleID`. In the case where a new participant needs to be created the following fields are supported by the Role Entity Data Processor.

- participantType - this is a code table entry from the `ConcernRoleType` code table. For example, use RL7 to create a new Prospect Person
- firstName
- middleInitial
- lastName
- SSN
- dateOfBirth
- lastName
- lastName

- street1
- city
- state
- zipCode

Updating Non Evidence Entities

Previous Sections have illustrated how it is possible to configure Life Events to automatically map updates through to Evidence Entities on multiple integrated cases. Sometimes Life Events will be required to update non-Evidence entities such as a Residential Address, Employment or some other customer specific non-Evidence entity. Typically these entities will be shared across multiple cases. It is also typical that these entities would not follow the same controlled Life Cycle as evidence entities. Evidence has many advantages:

- It is temporal
- It is case specific, with sharing of updates between cases being controlled through the Evidence Broker
- Case Workers can veto acceptance of updates that come from external sources like Universal Access
- It has an in-edit/approval cycle
- It has support for verifications

Non evidence entities have none of these advantages and safeguards. A decision by Analysts to update non Evidence entities based on Life Events should be made with due care, especially if the changes can be applied simultaneously across multiple cases. It is possible to update non Evidence entities but this will always involve custom code. It is strongly recommended that the design of such functionality includes safeguards to ensure that at least one Agency worker gets to manually approve the changes before they are applied to the system.

Configuring the Evidence Broker for use with the Holding Case:

The Holding Case is of little value by itself. It is only a holding area for Evidence before it is sent somewhere else. Normally, after data is updated on the Holding Case, the goal is to broker these updates to Integrated Cases so that caseworkers can evaluate the changes and apply them to the relevant cases.

For example, after the data is accepted onto the Integrated Cases, a user can see the positive impact of submitting a Life Event because the updated data has an impact on the user's benefits. The bridge between the Holding Case and the Integrated Cases is crossed only if the appropriate Evidence Broker configuration is defined. The following section demonstrates how that can be achieved. For more information, see the *Cúram Evidence Broker Developers Guide* on background on the Evidence Broker.

Configuring sharing from The Holding Case

An evidence configuration for sharing of Pregnancy evidence from the Holding Case to an Integrated Case is shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
  <table name="EVIDENCEBROKERCONFIG">
    <column name="EVIDENCEBROKERCONFIGID" type="id"/>
    <column name="SOURCETYPE" type="text" />
    <column name="SOURCEID" type="id" />
    <column name="TARGETTYPE" type="text" />
    <column name="TARGETID" type="id"/>
    <column name="SOURCEEVIDENCETYPE" type="text"/>
    <column name="TARGETEVIDENCETYPE" type="text"/>
    <column name="AUTOACCEPTIND" type="bool"/>
    <column name="WEBSERVICESIND" type="bool"/>
    <column name="SHAREDTYPE" type="text"/>
    <column name="RECORDSTATUS" type="text"/>
    <column name="VERSIONNO" type="number"/>
    <row>
      <attribute name="EVIDENCEBROKERCONFIGID">
        <value>10003</value>
      </attribute>
      <attribute name="SOURCETYPE">
        <value>CT10301</value>
      </attribute>
      <attribute name="SOURCEID">
        <value>10330</value>
      </attribute>
      <attribute name="TARGETTYPE">
        <value>CT5</value>
      </attribute>
      <attribute name="TARGETID">
        <value>4</value>
      </attribute>
      <attribute name="SOURCEEVIDENCETYPE">
        <value>ET10000</value>
      </attribute>
      <attribute name="TARGETEVIDENCETYPE">
        <value>ET10074</value>
      </attribute>
      <attribute name="AUTOACCEPTIND">
        <value>0</value>
      </attribute>
      <attribute name="WEBSERVICESIND">
        <value>0</value>
      </attribute>
      <attribute name="SHAREDTYPE">
        <value>SET2002</value>
      </attribute>
      <attribute name="RECORDSTATUS">
        <value>RST1</value>
      </attribute>
      <attribute name="VERSIONNO">
        <value>1</value>
      </attribute>
    </row>
  </table>
```

When evidence is shared from the Holding Case to another Integrated Case, the source type needs to be `CT10301` and the source ID needs to be set to `10330`. The source evidence type needs to be set to `ET10000`, which is the code for all Evidence that is stored in Holding Cases. Evidence of this type is known as `Holding Evidence`. The target evidence type in this case is `ET10074`. In Cúram Common Evidence, this particular designation identifies Pregnancy Evidence. The evidence sharing type needs to be set to `SET2002`, which is the code for Non-Identical Sharing.

Note: The `AUTOACCEPTIND` is set to `0`. It is recommended strongly that this value always be set to `0` when it is shared from a Holding Case to an Integrated Case. This setting means that a caseworker always gets to examine any changes that come in from the citizen's Holding Case
. Assuming the caseworker agrees with the changes, the **Incoming Evidence** link of the Integrated Case Evidence page can be used to synchronize the data from the Holding Case in the normal way.

To establish Evidence Broker Configuration for a custom component, a DMX file must be created that contains the configuration that follows the previous example, for example, `%SERVER_DIR%\components\Custom\data\initial\EBROKER_CONFIG.dmx`

In sharing Holding Evidence to a Standard Evidence Entity like a Pregnancy, the Evidence Broker copies the Holding Evidence that contains the Pregnancy data into a new Pregnancy Evidence Record in the target Integrated Case. Previous, it has been alluded that Holding Evidence is not standard Evidence. In fact, it is stored in an XML representation, so while the Holding Evidence is copied to the Target Evidence type, the Evidence Broker converts the XML data into standard Evidence data. To assist with this conversion process, it is necessary to supply metadata. An example of this metadata is illustrated in the following code block:

```
<?xml version="1.0" encoding="UTF-8"?>
<data-hub-config>
  <evidence-config package="curam.holdingcase.evidence">
    <entity name="HoldingEvidence" ev-type-code="ET10000">
      <attribute name="entityStruct">
        curam.citizen.datahub.holdingcase.holdingevidence.struct.
        +HoldingEvidenceDtls
      </attribute>
    </entity>
    <entity name="Pregnancy" ev-type-code="ET10074">
      <attribute name="entityStruct">
        curam.evidence.entity.struct.PregnancyDtls
      </attribute>
      <related-entity>
        <case-participant-role>
          <attribute name="linkAttribute">
            fahCaseParticipantRoleID
          </attribute>
        </case-participant-role>
        <case-participant-role>
          <attribute name="linkAttribute">
            caseParticipantRoleID
          </attribute>
        </case-participant-role>
      </related-entity>
    </entity>
  </evidence-config>
</data-hub-config>
```

The metadata describes each of the entities that can be copied from the Holding Case to an Integrated Case and vice versa. The metadata describes the `dtls` structs that are used to build the target evidence. It also describes which of the attributes in Case Evidence refer to case participant roles. This information ensures that when the Holding Evidence is copied, it does not blindly copy case participant role identifiers from Holding Evidence. Instead, it looks for the equivalent case participant role ID on the target case and, if it does not exist, creates one.

This metadata is stored in an `AppResource` resource store key. The resource store key is identified by the Cúram Environment Property `curam.workspaceservices.datahub.metadata`. The initially configured value for this variable defaults to the value `curam.workspaceservices.datahub.metadata`. This variable points to default Holding Evidence Data Hub metadata. The following steps can be used to replace the default Holding Evidence Data Hub metadata with a custom version to support all Evidence Types that need to be brokered from the Holding Case to all Integrated Cases:

- Copy the contents of `%SERVER_DIR%\components\WorkspaceServices\data\initial\clob\DataHubMetaData.xml` to `%SERVER_DIR%\components\Custom\data\initial\clob\CustomDataHubMetaData.xml`

- Edit the contents of `CustomDataHubMetaData.xml` to describe all the Evidence Entities that need to be updated by the Data Hub.
- Create a file `%SERVER_DIR%\components\Custom\data\initial\APP_RESOURCES.dmx`. Add an entry to this file as shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="APPRESOURCE">
<column name="resourceid" type="id" />
<column name="localeIdentifier" type="text"/>
<column name="name" type="text"/>
<column name="contentType" type="text"/>
<column name="contentDisposition" type="text"/>
<column name="content" type="blob"/>
<column name="internal" type="bool"/>
<column name="lastWritten" type="timestamp"/>
<column name="versionNo" type="number"/>
<row>
  <attribute name="resourceID">
    <value>10700</value>
  </attribute>
  <attribute name="localeIdentifier"> <value/>
  </attribute>
  <attribute name="name">
    <value>custom.datahub.metadata</value>
  </attribute>
  <attribute name="contentType">
    <value>text/plain</value>
  </attribute>
  <attribute name="contentDisposition"> <value>inline</value>
  </attribute> <
  attribute name="content"> <value> ./Custom/data/initial/clob/CustomDataHubMetaData.xml </value>
  </attribute> <attribute name="internal"> <value>0</value> </attribute>
  <attribute name="lastWritten"> <value>SYSTIME</value>
  </attribute> <attribute name="versionNo"> <value>1</value>
  </attribute>
</row>
</table>
```

- Create or append to the file `%SERVER_DIR%\components\Custom\properties\Environment.xml` adding an entry along the following lines:

```
<environment>
   <type name="dynamic_properties">
     <section code="WSSVCS"
       name="Workspace Services - Configuration">
       <variable name="curam.workspaceservices.datahub.metadata"
         value="custom.datahub.metadata" onlyin="all"
         type="STRING">
         <comment>
           Identifies an AppResource used to configure DataHub
           meta-data.
         </comment>
       </variable>
     </section>
   </type>
 </environment>
```

Round Tripping and Configuring Sharing to The Holding Case

The previous section described how data is shared from the Holding Case to Integrated Cases. Analysts also might want to consider whether evidence needs to be transferred in the opposite direction - that is, from the Integrated Cases to the Holding Case. When sharing is configured from the Integrated Case to the Holding Case, changes made by the caseworker to selected evidence can be propagated back to the Holding Case. This process is essential for Life Events that need to pre-populate data from Evidence Entities in existing Integrated Cases. The example that follows shows how to configure Pregnancy Evidence for Sharing to the holding case.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="EVIDENCEBROKERCONFIG">
  <column name="EVIDENCEBROKERCONFIGID" type="id"/>
  <column name="SOURCETYPE" type="text" />
  <column name="SOURCEID" type="id" />
  <column name="TARGETTYPE" type="text" />
  <column name="TARGETID" type="id"/>
  <column name="SOURCEEVIDENCETYPE" type="text"/>
  <column name="TARGETEVIDENCETYPE" type="text"/>
  <column name="AUTOACCEPTIND" type="bool"/>
  <column name="WEBSERVICESIND" type="bool"/>
  <column name="SHAREDTYPE" type="text"/>
  <column name="RECORDSTATUS" type="text"/>
  <column name="VERSIONNO" type="number"/>
  <row>
    <attribute name="EVIDENCEBROKERCONFIGID">
      <value>2</value>
    </attribute>
    <attribute name="SOURCETYPE">
      <value>CT5</value>
    </attribute>
    <attribute name="SOURCEID">
      <value>4</value>
    </attribute>
    <attribute name="TARGETTYPE">
      <value>CT10301</value>
    </attribute>
    <attribute name="TARGETID">
      <value>10330</value>
    </attribute>
    <attribute name="SOURCEEVIDENCETYPE">
      <value>ET10074</value>
    </attribute>
    <attribute name="TARGETEVIDENCETYPE">
      <value>ET10000</value>
    </attribute>
    <attribute name="AUTOACCEPTIND">
      <value>1</value>
    </attribute>
    <attribute name="WEBSERVICESIND">
      <value>0</value>
    </attribute>
    <attribute name="SHAREDTYPE">
      <value>SET2002</value>
    </attribute>
    <attribute name="RECORDSTATUS">
      <value>RST1</value>
    </attribute>
    <attribute name="VERSIONNO">
      <value>1</value>
    </attribute>
  </row>
</table>
```

Note: Unlike Sharing from Holding Case to Integrated Case, the `AUTOACCEPTIND` is set to 1. This designation is because the target case is a Holding Case and Holding Cases are designed to operate unattended. It is not expected that caseworkers need to review items that are being shared onto the Holding Case as they come from an authoritative source, for instance, the Integrated Case.

Issues for consideration

With suitable configuration, It is possible to share data from the Holding Case to many different Integrated Cases. Take the example of two different Integrated Cases (cases A and B) that are configured to share information with a citizen's Holding Case (case H). Both cases A and B separately recorded an Income Evidence record for the citizen. In the citizen's Holding Case, this evidence record shows up as two separate Income Records. As far as cases A and B are concerned, they are two entirely separate records - A's view of the citizen's Income and B's view of the citizen's Income. However, to the citizen, this breakdown might not make much sense. The citizen has only one Income and is using one Portal to communicate with the Social Enterprise Management (SEM) agency or agencies concerned. Why does the citizen see two records for the same Income? In cases where there is sharing to multiple Integrated Cases from a single Holding Case, consideration needs to be given to creating another set of sharing relationships to be established from A to B and B to A. This consideration is an issue that requires proper consideration early on in the project lifecycle.

Putting it all Together: Previous topics have discussed how to create all the constituent pieces of a Life Event, this topic discusses how to join all these pieces together to make a completed Life Event. New Life Events can be configured using the Life Event Administration pages. Please refer to the *IBM Cúram Universal Access Guide* for more information on how to do this. Using the Administration Pages it is possible to create new Life Event Types and Life Event Channels, add rich text descriptions and associate the Life Events with IEG Scripts and Recommendation Rule Sets. Once all of the required Entities have been created in the Administration screens, the data can be extracted into a set of DMX files that can be used as a basis for ongoing development. The following set of commands can be used to extract the relevant dmx files:

```
build extractdata -Dtablename=LifeEventType
build extractdata -Dtablename=LifeEventContext
build extractdata -Dtablename=LifeEventCategory
build extractdata -Dtablename=LifeEventCategoryLink
build extractdata -Dtablename=LocalizableText
build extractdata -Dtablename=TextTranslation
```

The LocalizableText and TextTranslation tables contain all of the Life Event descriptions but they will also be filled with text translations that do not relate to Life Events. Developers should audit these DMX files removing any entries that do not correspond to the relevant Life Event descriptions before copying the dmx files to `%SERVER_DIR%\components\Custom\data\initial\`.

Life Events API Guide

A description of how to use the Java API for Life Events and the Citizen Data Hub.

Event APIs for Life Events

The Life Event Broker is instrumented with Guice events. Developers can write listeners that can be bound to these events. The available events are:

- `PreCreateLifeEvent` - Invoked before launching a Life Event
- `PostCreateLifeEvent` - Invoked after the Life Event has been initialized. That is after the Data Hub Transform and View Processors have been executed.
- `PreSubmitLifeEvent` - Invoked after the Life Event has been submitted but before the Update Processors have been run.
- `PostSubmitLifeEvent` - Invoked after the Life Event has been submitted.

Note that both the Pre and Post SubmitLifeEvent events are executed from within a Deferred Process so the current user is expected to be SYSTEM. Life Event Events should never attempt to change the contents of the Life Event. The code extract below shows how a Listener class, MyPreCreateListener can be bound to one of these Life Events:

```
Multibinder<LifeEventEvents.PreCreateLifeEvent>
   preCreateBinder =
    Multibinder.newSetBinder(binder(),
      new TypeLiteral<LifeEventEvents.PreCreateLifeEvent>() { /**/
    });

  preCreateBinder.addBinding().to(MyPreCreateListener.class);
```

Universal Access Web Services

A description of Universal Access web services with some sample SOAP requests, and how to develop peer code to communicate with those web services. In some scenarios, customers will deploy Universal Access to handle interactions with clients over the Internet, but will use an existing legacy system for case processing. To cater for these scenarios, Universal Access can be configured to communicate with various remote systems using web services.

Inbound and outbound web services

Universal Access supports the following outbound web services:

- Send an application for benefits.
- Withdraw an application for benefits.
- Send a Life Event.

Universal Access supports the following inbound web services:

- Create a citizen account on Universal Access.
- Link a user to a remote system (gives them the right to send information to those systems and receive information from them in turn).
- Unlink a user from a remote system.
- Receive an update to the status of a submitted application.
- Receive an update to the status of a request to withdraw an application.
- Receive a citizen message (for display on a citizen account home page).
- Receive payment information.
- Receive case contact information.

Web Services Security Considerations

Universal Access is designed to communicate with an arbitrary number of remote systems. These may be configured through the remote systems configuration page in the Cúram Administrator application.

Remote systems can invoke web services on Universal Access and must supply username/password credentials as part of the SOAP header, details of how to do this are described using sample web service requests. It is strongly recommended that a different username and password be assigned to each remote system. The username associated with a remote system is set in the Source User Name field of the remote system configuration page. Having a different user name for each remote system allows Universal Access to perform proper data-based security checks on the incoming service requests. This prevents one remote system sending requests to update data that is properly the concern of a different remote system.

Process Application Service

The Process Application Service web services.

Receive Application

This outbound web service is started by Universal Access (UA) on remote systems. It is used to communicate an application for benefits for one or more social programs. The Web Service Description Language (WSDL) describing this service can be found in `<CURAM_DIR>\EJBServer\components\WorkspaceServices\axis\ProcessApplicationService\ProcessApplicationService.wsdl`.

A web service request of this type contains the following information:

- `intakeApplicationType` - An ID that uniquely identifies an Intake Application Type.
- `applicationReference` – A unique reference for a particular application. This reference is a human-readable ID that is displayed to clients after they complete an application; for example, `512` or `756`. The application reference is used as an argument to other web services and needs to be stored by the receiver.
- `applicationLocale` – Denotes the preferred locale of the user who entered the application, for example `en_US`. This information needs to be stored by the receiver. Remote systems can send various information back to the client's account. Some of this information must be localized by the sender to the preferred locale of the client.
- `submittedDateTime` – The date and time at which the application was submitted. This information is in XML schema `dateTime` format, for example, 2012-05-29T15:34:49.000+01:00.
- `programsAppliedFor` – This field contains a list of the programs that were applied for as part of this application. Each program is referred to by a unique reference. This information corresponds to the value of the Reference field configured in the Programs section of UA configuration. For example:

  ```
  <ns1:programsAppliedFor>
      <ns1:programTypeReference>CashAssistance</ns1:programTypeReference>
      <ns1:programTypeReference>SNAP</ns1:programTypeReference>
  </ns1:programsAppliedFor>
  ```
- `applicationData` – Contains a base64 encoded representation of the intake data. This intake data is the XML representation of the XML data store associated with an application.
- `applicationSchemaName` – The name of the schema that is used to create the data store for the application.
- `senderIdentification` – Identifies the sender of the request. The sender identification contains two parts, 1) the identifier of the system from which the request originates, 2) The Citizen Workspace Account ID of the user that created the request. The second part is optional, applications submitted anonymously do not contain part two but applications that are submitted by a logged in user do.
- `supplementaryInformation` – optional, reserved for future use.

The receiver of this information is expected to record the details of the application keyed against sender identification and intake application reference.

On success, the implementation of this web service must return the Boolean value `true` to indicate that the request was processed successfully. In the case that a problem occurs in processing the request, a fault must be returned containing a string to indicate the nature of the problem. The String needs to be localized to the locale of the UA Server since it appears in the server log files.

Note: The receiver can receive multiple applications with the same Intake Application reference but the intake application reference is always unique for a particular sender. For example, Systems A and B send a `receiveApplication()` request to system X. Both requests have the `applicationReference` 256.

Note: The receiver never should receive two applications from A with an application reference of 256.

Receive Withdrawal Request

This outbound web service is invoked by Universal Access on remote systems. It is used by clients to withdraw an application that they have previously submitted using the Receive Application Service. WSDL describing this service can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\axis\ProcessApplicationService\ProcessApplicationService.wsdl. A web service request of this type contains the following information:

- applicationReference – A unique reference for the application to be withdrawn. This refers to the id transmitted with the Receive Application service request.
- programTypeReference – A reference that identifies the program being withdrawn. Each program type is referred to by a unique reference. This corresponds to the value of the Reference field configured in the Programs section of Universal Access configuration. For example "CashAssistance".
- requestSubmittedDateTime – A timestamp indicating when the request was submitted in XML Schema dateTime format. For example, 2012-05-29T15:34:49.000+01:00
- withdrawalRequestReason – The value is taken from the code table WithdrawalRequestReason. Values for this code table are
  - WRES1001 – Attained employment
  - WRES1002 – Change of circumstances
  - WRES1003 – Filed in error
- withdrawalRequestID – An id that uniquely identifies this withdrawal request from the sending instance of Universal Access.
- senderIdentification – Identifies the sender of the request. The sender identification contains two parts, 1) the identifier of the system from which the request originates, 2) The Citizen Workspace Account ID of the user that created the request.
- supplementaryInformation – optional, reserved for future use.

The expected result following successful processing is a receiveWithdrawalRequestResponse as follows:

```
<receiveWithdrawalRequestResponse>
  <result>true</result>
</receiveWithdrawalRequestResponse>
```

The service implementation should return a fault if there is an error processing the request. The fault string should be globalized to the locale of the Universal Access Server since it will appear in the server log files. Some problems that may arise include:

- A withdrawal request with the given ID has already been sent by the given instance of Universal Access.
- The application reference referred to is not recognized as an application previously transmitted in a Receive Application service invocation from the same Universal Access instance.

The withdrawal request application is processed by the receiving agency after which a response should be sent in the form of a withdrawal request update. See the sample SOAP request for this web service.

Update Application Service

The Update Application Service web services.

Intake Program Application Update

This is an inbound web service invoked by remote systems on Universal Access. It is used to inform the Universal Access System of changes to the status of an application for benefits that was previously received via the Receive Application web service. The status of an application can transition to Approved, Denied or Withdrawn. Where an application is denied a reason can be included in the web service message. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\UpdateApplication.xsd. See the sample SOAP request for this web service.

A web service request of this type contains the following information:

- curamReferenceID – This must match the applicationReference element for the corresponding Receive Application request.
- programApplicationStatus – This can take the following values:
  - IPAS1002 – Withdrawn
  - IPAS1003 – Approved
  - IPAS1004 – Denied
- programApplicationDisposedDateTime – This is a formatted date time string in the standard IBM Cúram ISO8601 format – "YYYYMMDD HH:MM:SS".
- programApplicationDenialReason – Optional, if the status sent is IPAS1004, this contains free text describing the reason for denial. The denial reason should be taken from the code table IBM Cúram IntakeProgApplDenyReason.

The web service request needs to be sent with a Cúram security credential (see a sample SOAP message for details). The user name placed within the credential must match the Source User Name entered into the Remote System entry corresponding to the peer system sending the request.

Withdrawal Request Update

This is an inbound web service invoked by remote systems on Universal Access. It is used to inform the Universal Access System of changes to the status of a Withdrawal Request that was previously submitted using the Receive Withdrawal Request web service. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\UpdateApplication.xsd. See the sample SOAP request for this web service.

A web service request of this type contains the following information:

- curamReferenceID – This must match the withdrawalRequestID in the corresponding Receive Withdrawal Request message.
- withdrawalRequestStatus – This an enumeration taking the following values:
  - WREQ1002 – Approved
  - WREQ1003 – Denied
- resolvedDateTime – A time stamp in the standard IBM Cúram ISO8601 format – "YYYYMMDD HH:MM:SS".

- withdrawalRequestDenialReason – Optional. In the case there the withdrawal request was denied, a textual explanation for the denial. The sender must localize this to the locale of the client who originally submitted the application.

See the sample SOAP request for the Withdrawal Request Update operation.

On success this operation returns a document indicating that the request has succeeded. On failure, a fault is raised. Reasons for failure include:
- The withdrawal request id does not match a known withdrawal request id.
- The withdrawal request state transition is invalid.

Life Event Service

This outbound web service is invoked by Universal Access on remote systems. WSDL describing this service can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\axis\LifeEventService\LifeEvent.wsdl.

A request for this web service contains the following fields:
- lifeEventReference – Describes the type of the Life Event, for example "Change of Address"
- senderIdentification – Identifies the sender of the request. The sender identification contains two parts, 1) the identifier of the system from which the request originates, 2) The Citizen Workspace Account ID of the user that created the request.
- lifeEventData - Contains a base64 encoded representation of the Life Event data. This Life Event data is the XML representation of the XML datastore associated with an Life Event.
- lifeEventSchemaName – The name of the schema used to create the data store for the Life Event.
- submittedDateTime – The date and time when the Life Event was submitted. An XML Schema dateTime. For example, 2012-05-29T15:34:49.000+01:00
- supplementaryInformation – optional, reserved for future use.

The implementation should return a response of type lifeEventResponse with the content "true" when the Life Event is successfully processed. If there is an error processing the Life Event then the system should return a fault in accordance with the WSDL specification.

Create Account Service

This is an inbound web service invoked by remote systems on Universal Access. It is used to create a Citizen Workspace Account for users who previously submitted an Intake Application anonymously. The service actually performs two discrete functions:
- Create an account for a previously anonymous user.
- Link that account to the remote system that is invoking the Create Account Web Service.

If a Citizen Workspace user is "linked" to a remote system, it means that user is registered on the remote system and the remote system will recognize requests from that Citizen Workspace user as relating to a particular case, cases or an individual on the remote system. This has serious security implications on the remote system – The remote system sending a request to link a user or create an account for a user must be convinced of the identity of the individual who owns the account. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\ExternalAccountCreate.xsd. See the sample SOAP request for this web service.

A create account request contains the following information:

- firstName – The client's first name.
- middleName – The client's middle name. Optional.
- surname – The client's last name.
- username – The username for the newly created account.
- password – The password for the newly created account.
- confirmPassword – Confirmation of the password. Must match password.
- secretQuestionType – The type of secret question selected to unlock the user's account. Values should correspond to entries from the SecretQuestionType code table. For example, SQT1 – Mother's maiden name.
- answer – An answer to the secret question. Non empty.
- termsAndConditionsAccepted – Boolean indication that the client has accepted the terms and conditions on which the account is created.
- intakeApplicationReference – Refers to the unique applicationReference passed in as part of the receive application request. If this is specified, a link will be created between the application and the newly created account.
- clientIDOnRemoteSystem – This is a unique identifier that can be used to identify the user of this account on the remote system. There is no prescribed form for this id, it could be a Social Security Number for example. It must be capable of uniquely identifying the client on the remote system.
- sourceSystem – Identifies the remote system that sent this request. This must match the name of a remote system configured in the administration application. For more information about configuring remote systems, see "Configuring Remote Systems" in the `IBM Cúram Universal Access Configuration Guide`.

If successful this returns the id of the created citizen workspace account. Problems that occur during the processing of the request are flagged by a fault response. Possible issues include:

- An account has already been associated with the intake application reference.
- The username already exists.
- The user name or password do not meet minimum mandatory criteria such as password strength, user name length.

Link Service

This is an inbound web service invoked by remote systems on Universal Access. It is used to link a Citizen Workspace Account to a remote system. See the section on Create Account Service for a general discussion of the implications of linking a user. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\ExternalAccountLink.xsd. See the sample SOAP request for this web service.

This web service request contains the following information:

- sourceSystem – The name of the remote system sending the request. Must match the name of a remote system configured in the system.
- citizenWorkspaceAccountID – The unique citizen workspace account id.

- clientIDOnRemoteSystem - This is a unique identifier that can be used to identify the user of this account on the remote system. There is no prescribed form for this id, it could be a Social Security Number for example. It must be capable of uniquely identifying the client on the remote system.
- createdByUsername – The username on the remote system responsible for this request.

On success this operation returns a document indicating that the request has succeeded. On failure, a fault is raised. Reasons for failure include:

- The citizen workspace account id is invalid, does not exist or is associated with a de-activated account.
- The citizen workspace account in question is already linked to this remote system.

Unlink Service

This is an inbound web service invoked by remote systems on Universal Access. It is used to unlink a Citizen Workspace Account from a remote system. After executing this service it will not be possible for the user of the unlinked account to submit Life Events to this remote system, for example. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\ExternalAccountUnlink.xsd. See the sample SOAP request for this web service.

This web service request contains the following information:

- sourceSystem – The name of the remote system sending the request.
- citizenWorkspaceAccountID – The unique ID of the Citizen Workspace Account being unlinked.

On success this operation returns a document indicating that the request has succeeded. On failure, a fault is raised. Reasons for failure include:

- The indicated account does not exist or is not active.
- The indicated account is not linked to the remote system sending the request.

Citizen Message

This is an inbound web service invoked by remote systems on Universal Access. It is used to send Citizen Messages that are displayed on a user's Home Page when they log in to the Citizen Account. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\ExternalCitizenMessage.xsd. See the sample SOAP request for this web service.

This web service request contains the following information:

- sourceSystem – The name of the remote system sending the request.
- citizenWorkspaceAccountID – The unique citizen workspace account id.
- cityIndustryType – Denotes the type of industry associated with the message. The values for this element must match codes from the CityIndustry code table.
- relatedID – Refers to the id of an underlying entity in the remote system to which the message refers. For example, if the message concerns a payment then the related ID identifies the ID of the payment within the remote system.
- externalCitizenMessageType – The external citizen message type, taken from the ExternalCitizenMessageType codetable.

- messageTitle – The title of the message. It is the responsibility of the remote system to localize this to the locale of the end user.
- messageBody – The body of the message. It is the responsibility of the remote system to localize this to the locale of the end user.
- effectiveDate – Optional. The date from which the message is effective. It will only be displayed from this date onwards. The date must be in the format – "YYYY-MM-DD". If an effective date is not provided then the current date is taken as the effective date.
- expiryDate – The date that the message is set to expire. Following this date, the message will not be displayed to the user. The date must be in the format – "YYYY-MM-DD".
- priority – A boolean value to indicate whether this message is a high priority.

Some messages are designed such that a newer message can replace an older one. For example, a message is sent concerning a meeting. The time of the meeting changes and a new message is sent with the updated time for the meeting. The client does not see both messages, rather the second message replaces the first and only the second message is seen. One external message will automatically replace another external message if the following fields match those of an existing message: sourceSystem, externalCitizenMessageType and relatedID.

Payment Service

This is an inbound web service invoked by remote systems on Universal Access. This service is used to transmit information about one or more payments. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\ExternalPayment.xsd. See the sample SOAP request for this web service.

This web service request can contain one or more Payments. This allows the remote system to batch up payments and send them as a single request for performance reasons. Each payment can relate to an entirely separate Universal Access account. A single payment may contain a payment breakdown. A payment breakdown may contain one or more payment line items.

A single Payment contains the following information:

- paymentID – Together with the source system, this uniquely identifies a payment.
- sourceSystem – The name of the remote system sending the request. Must match the name of a remote system configured in the system.
- citizenWorkspaceAccountID – The unique citizen workspace account id.
- cityIndustryType – Denotes the type of industry associated with the payment. The values for this element must match codes from the CityIndustry code table. Optional.
- paymentAmount – The headline value for the payment as a whole. This payment may optionally be further broken into a number of line items.
- currency – The currency in which the payment was made, contains values from the Currency code table. Optional.
- paymentMethod – The method by which the payment was made, contains values from the MethodOfDelivery code table.
- paymentStatus – The status of the payment, for example cancelled, processed, suspended etc. Contains values from PmtReconciliationStatus code table.
- effectiveDate – The effective date of the payment in the format "YYYY-MM-DD".

- coverPeriodFrom – The start date of the period covered by this payment. In the format "YYYY-MM-DD".
- coverPeriodTo – The end date of the period covered by this payment. In the format "YYYY-MM-DD".
- dueDate – The date that the payment was due to be paid. In the format "YYYY-MM-DD".
- payeeName – The name of the payee for this payment.
- payeeAddress – The address that the payment was sent to (in the case of a cheque). Optional.
- paymentReferenceNo – Uniquely identifies a payment within a given remote system.
- bankSortCode - The sort code of the bank account to which this payment is delivered.
- bankAccountNo – The bank account number to which payment is made.
- A payment may contain a Payment Breakdown (optional).

A Payment Breakdown contains one or more Payment Line Items. A Payment Line Item contains the following information:

- caseName – The human readable name of the case on the remote system with which this payment is associated.
- The case name must be localised to the locale of the client. This case name must match the case name displayed on the Contact Information page.
- caseReference – This uniquely identifies the case on a given remote system.
- componentType – This contains a code from the FinComponentType code table.
- debitAmount – The amount debited if this payment was a debit.
- creditAmount – The amount credited if this payment was a credit.
- coverPeriodFrom - The start date of the period covered by this payment. In the format "YYYY-MM-DD".
- coverPeriodTo – The end date of the period covered by this payment. In the format "YYYY-MM-DD".

It is important to note that payments can supersede previously submitted payments. For example, a payment is submitted from TestSystem with paymentID 1234. Subsequently another payment arrives from TestSystem with the same paymentID, 1234. This payment replaces the previous payment. The previous payment is physically removed along with all its related payment line items. A typical example of where this might occur is when a previously issued payment is cancelled.

Contact Service

This is an inbound web service invoked by remote systems on Universal Access. This service is used to update a register of case worker contact details relating to a remote system. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\ExternalContact.xsd. See the sample SOAP request for this web service.

A contact web service request contains the following information:

- sourceSystem – The name of the remote system sending the request. Must match the name of a remote system configured in the system.

- contactReference – A reference for the contact, unique within the source remote system.
- fullName – The full name of the case worker.
- phoneNumber – The phone number of the case worker. Optional.
- mobilePhoneNumber – The mobile/cell phone number of the case worker. Optional.
- faxNumber – The fax number for the case worker. Optional.
- email – The email address of the case worker. Optional.

If a request is received with the same source system and contact reference as a pre-existing entry then the information in the newer request supersedes the pre-existing information.

Case Service

This is an inbound web service invoked by remote systems on Universal Access. This service is used to update details of cases associated with a particular Citizen Account. The schema for the payload of web service requests of this type can be found in <CURAM_DIR>\EJBServer\components\WorkspaceServices\webservices\ExternalCase.xsd. See the sample SOAP request for this web service.

A web service request of this type contains the following information:

- sourceSystem – The name of the remote system sending the request. Must match the name of a remote system configured in the system.
- contactReference – A reference for the contact, unique within the source remote system, this must match a contact reference previously transmitted via a Contact Service request.
- caseReference – This is a case reference and must be unique within the remote system that is the source of this request.
- caseName - The human readable name of the case on the remote system. The case name must be localised to the locale of the client. Case names used in the Payment web service should match case names provided in this request.
- citizenWorkspaceAccountID – The unique citizen workspace account id.

If a request is received with the same source system and case reference as a pre-existing entry then the information in the newer request supersedes the pre-existing information.

Sample SOAP Requests

A list of sample SOAP requests to help with development.

Intake Program Application Update

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices
.curam" xmlns:xsd="http://dom.w3c.org/xsd">
   <soapenv:Header>
     <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
     <Username>userforpeersystem</Username>
     <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:updateIntakeProgramApplication>
      <rem:xmlMessage>
      <intakeProgramApplicationUpdate>
        <applicationReference>256</applicationReference>
        <applicationProgramReference>joannesprogram
</applicationProgramReference>
        <programApplicationStatus>IPAS1004</programApplicationStatus>
        <programApplicationDisposedDateTime>
          20120528 17:19:47
        </programApplicationDisposedDateTime>
        <programApplicationDenialReason>IPADR1001
</programApplicationDenialReason>
      </intakeProgramApplicationUpdate>
         </rem:xmlMessage>
      </rem:updateIntakeProgramApplication>
   </soapenv:Body>
</soapenv:Envelope>
```

Withdrawal Request Update

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

  <column name="
      searchServiceFieldId
      " type="text" />
  <column name="
      searchServiceId
      " type="text" />
  <column name="
      name
      " type="text" />
  <column name="
      indexed
      " type="bool" />
  <column name="
      type
      " type="text" />
  <column name="
      stored
      " type="bool" />
  <column name="
      entityName
      " type="text" />
  <column name="
      analyzerName
      " type="text" />
  <column name="
      untokenized
      " type="bool" />

  <row>
    <attribute name="searchServiceFieldId">
      <value>
      field0
      </value>
    </attribute>
    <attribute name="searchServiceId">
      <value>
      PersonSearch
      </value>
    </attribute><attribute name="name">
      <value>
      primaryAlternateID
      </value>
    </attribute><attribute name="indexed"> <soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:rem="http://remote.externalservices.workspaceservices.curam"
xmlns:xsd="http://dom.w3c.org/xsd">
   <soapenv:Header>
     <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
     <Username>userforpeersystem</Username>
     <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:updateWithdrawalRequest>
         <rem:xmlMessage>
    <withdrawalRequestUpdate>
      <curamReferenceID>-6897262829317914624</curamReferenceID>
      <withdrawalRequestStatus>WREQ1002</withdrawalRequestStatus>
      <resolvedDateTime>20120525 11:30:50</resolvedDateTime>
    </withdrawalRequestUpdate>
         </rem:xmlMessage>
      </rem:updateWithdrawalRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Create Account

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices.
curam" xmlns:xsd="http://dom.w3c.org/xsd">
   <soapenv:Header>
     <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
     <Username>admin</Username>
     <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:createAccount>
         <!--Optional:-->
         <rem:xmlMessage>
            <!--Optional:-->
     <cre:AccountCreate xmlns:cre="http://www.curamsoftware.com/
WorkspaceServices/ExternalAccountCreate">
        <firstName>John</firstName>
        <middleName>M</middleName>
        <surname>Doe</surname>
        <username>johnmdoe</username>
        <password>password1</password>
        <confirmPassword>password1</confirmPassword>
        <secretQuestionType>SQT1</secretQuestionType>
        <answer>mypassword1</answer>
        <termsAndConditionsAccepted>true</termsAndConditionsAccepted>
        <intakeApplicationReference>256</intakeApplicationReference>
        <clientIDOnRemoteSystem>112233445566</clientIDOnRemoteSystem>
        <sourceSystem>TestSystem</sourceSystem>
    </cre:AccountCreate>
         </rem:xmlMessage>
      </rem:createAccount>
   </soapenv:Body>
</soapenv:Envelope>
```

Account Link

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices.
curam" xmlns:xsd="http://dom.w3c.org/xsd">
   <soapenv:Header>
     <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
     <Username>admin</Username>
     <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:linkTargetSystemToAccount>
         <rem:xmlMessage>
          <lnk:AccountLink xmlns:lnk="http://www.curamsoftware.com/
WorkspaceServices/ExternalAccountLink">
```

```
           <sourceSystem>TestSystem</sourceSystem>
           <citizenWorkspaceAccountID>7081910414040104960
</citizenWorkspaceAccountID>
           <clientIDOnRemoteSystem>112233445566</clientIDOnRemoteSystem>
          <createdByUsername>testuser</createdByUsername>
         </lnk:AccountLink>
         </rem:xmlMessage>
      </rem:linkTargetSystemToAccount>
   </soapenv:Body>
</soapenv:Envelope>
```

Account UnLink

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices.
curam" xmlns:xsd="http://dom.w3c.org/xsd">
   <soapenv:Header>
     <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
     <Username>admin</Username>
     <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:unlinkTargetSystemFromAccount>
         <!--Optional:-->
         <rem:xmlMessage>
         <unl:AccountUnlink xmlns:unl="http://www.curamsoftware.com/
WorkspaceServices/ExternalAccountUnlink">
           <sourceSystem>TestSystem</sourceSystem>
           <citizenWorkspaceAccountID>7081910414040104960
</citizenWorkspaceAccountID>
         </unl:AccountUnlink>
         </rem:xmlMessage>
      </rem:unlinkTargetSystemFromAccount>
   </soapenv:Body>
</soapenv:Envelope>
```

Citizen Message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices.
curam" xmlns:xsd="http://dom.w3c.org/xsd">
    <soapenv:Header>
  <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
     <Username>admin</Username>
     <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:createMessage>
         <rem:xmlMessage>
<cm:CitizenMessage xmlns:cm="http://www.curamsoftware.com/
WorkspaceServices/ExternalCitizenMessage">
  <sourceSystem>TestSystem</sourceSystem>
  <cityIndustryType>CMI9001</cityIndustryType>
  <citizenWorkspaceAccountID>7081910414040104960
</citizenWorkspaceAccountID>
  <relatedID>6060</relatedID>
  <externalCitizenMessageType>PMT2004</externalCitizenMessageType>
  <messageTitle>Hello, World!</messageTitle>
  <messageBody>This is the body of the message.</messageBody>
  <effectiveDate>2000-01-01</effectiveDate>
  <expiryDate>2020-01-01</expiryDate>
  <priority>false</priority>

</cm:CitizenMessage>
```

```
         </rem:xmlMessage>
      </rem:createMessage>
   </soapenv:Body>
</soapenv:Envelope>
```

Payment (Simple)

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices.
curam" xmlns:xsd="http://dom.w3c.org/xsd">
   <soapenv:Header>
     <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
      <Username>admin</Username>
      <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:create>
         <rem:xmlMessage>
     <tns:Payment xmlns:tns="http://www.curamsoftware.com/
WorkspaceServices/ExternalPayment">
        <paymentID>1554</paymentID>
        <sourceSystem>TestSystem</sourceSystem>
        <cityIndustryType>CMI9001</cityIndustryType>
        <citizenWorkspaceAccountID>7081910414040104960
</citizenWorkspaceAccountID>
        <paymentAmount>50.00</paymentAmount>
        <currency>EUR</currency>
        <paymentMethod>CHQ</paymentMethod>
        <paymentStatus>PRO</paymentStatus>
        <effectiveDate>2012-01-01</effectiveDate>
        <coverPeriodFrom>2012-01-01</coverPeriodFrom>
        <coverPeriodTo>2012-01-01</coverPeriodTo>
        <dueDate>2012-01-01</dueDate>
        <payeeName>Dorothy</payeeName>
        <payeeAddress>12 Gloster St., WA 6008</payeeAddress>
        <paymentReferenceNo>F</paymentReferenceNo>
        <bankSortCode>933384</bankSortCode>
        <bankAccountNo>88776655</bankAccountNo>
        <PaymentBreakdown>
           <PaymentLineItem>
                <caseName>I</caseName>
                <caseReferenceNo>J</caseReferenceNo>
                <componentType>C10</componentType>
                <debitAmount>22.45</debitAmount>
                <creditAmount>50.76</creditAmount>
                <coverPeriodFrom>2012-01-01</coverPeriodFrom>
                <coverPeriodTo>2012-01-01</coverPeriodTo>
            </PaymentLineItem>
        </PaymentBreakdown>
    </tns:Payment>
  </rem:xmlMessage>
</rem:create>
</soapenv:Body>
</soapenv:Envelope>
```

Payment (Batched)

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices.
curam" xmlns:xsd="http://dom.w3c.org/xsd">
   <soapenv:Header>
     <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
      <Username>admin</Username>
      <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
```

```
      <rem:create>
         <rem:xmlMessage>
     <tns:Payments xmlns:tns="http://www.curamsoftware.com/
WorkspaceServices/ExternalPayment">
        <Payment>
            <paymentID>2346</paymentID>
            <sourceSystem>TestSystem</sourceSystem>
            <cityIndustryType>CMI9001</cityIndustryType>
            <citizenWorkspaceAccountID>8306889512684879872
</citizenWorkspaceAccountID>
            <paymentAmount>48.00</paymentAmount>
            <currency>EUR</currency>
            <paymentMethod>CHQ</paymentMethod>
            <paymentStatus>PRO</paymentStatus>
            <effectiveDate>2012-01-01</effectiveDate>
            <coverPeriodFrom>2012-01-01</coverPeriodFrom>
            <coverPeriodTo>2012-01-01</coverPeriodTo>
            <dueDate>2012-01-01</dueDate>
            <payeeName>D</payeeName>
            <payeeAddress>E</payeeAddress>
            <paymentReferenceNo>F</paymentReferenceNo>
            <bankSortCode>G</bankSortCode>
            <bankAccountNo>H</bankAccountNo>
            <PaymentBreakdown>
                <PaymentLineItem>
                    <caseName>I</caseName>
                    <caseReferenceNo>J</caseReferenceNo>
                    <componentType>C24000</componentType>
                    <debitAmount>22.45</debitAmount>
                    <creditAmount>49.76</creditAmount>
                    <coverPeriodFrom>2012-01-01</coverPeriodFrom>
                    <coverPeriodTo>2012-01-01</coverPeriodTo>
                </PaymentLineItem>
                <PaymentLineItem>
                    <caseName>I</caseName>
                    <caseReferenceNo>J</caseReferenceNo>
                    <componentType>C24000</componentType>
                    <debitAmount>22.45</debitAmount>
                    <creditAmount>49.76</creditAmount>
                    <coverPeriodFrom>2012-01-01</coverPeriodFrom>
                    <coverPeriodTo>2012-01-01</coverPeriodTo>
                </PaymentLineItem>
            </PaymentBreakdown>
        </Payment>
    </tns:Payments>
         </rem:xmlMessage>
      </rem:create>
   </soapenv:Body>
</soapenv:Envelope>
```

Contact

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices.
curam" xmlns:xsd="http://dom.w3c.org/xsd">
   <soapenv:Header>
  <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
     <Username>admin</Username>
     <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:updateExternalContact>
         <rem:xmlMessage>
     <con:ContactInfo xmlns:con="http://www.curamsoftware.com/
WorkspaceServices/ExternalContact">
        <sourceSystem>TestSystem</sourceSystem>
        <contactReference>CON_100</contactReference>
```

```
          <fullName>Harry Neilan</fullName>
         <phoneNumber>1-800-CALL-ME</phoneNumber>
         <mobilePhoneNumber>1-800-CALL-MOB</mobilePhoneNumber>
         <faxNumber>1-800-CALL-FAX</faxNumber>
         <email>harry@x.org</email>
    </con:ContactInfo>
          </rem:xmlMessage>
      </rem:updateExternalContact>
   </soapenv:Body>
</soapenv:Envelope>
```

Cases

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:rem="http://remote.externalservices.workspaceservices.
curam" xmlns:xsd="http://dom.w3c.org/xsd">
<soapenv:Header>
     <curam:Credentials xmlns:curam="http://www.curamsoftware.com">
     <Username>admin</Username>
     <Password>password</Password>
   </curam:Credentials>
   </soapenv:Header>
   <soapenv:Body>
      <rem:updateExternalCase>
         <rem:xmlMessage>
    <cas:CaseInfo xmlns:cas="http://www.curamsoftware.com/
WorkspaceServices/ExternalCase">
        <sourceSystem>TestSystem</sourceSystem>
        <contactReference>CON_100</contactReference>
        <caseReference>CAS_109</caseReference>
        <caseName>My Benefit Case - 103</caseName>
        <citizenWorkspaceAccountID>8306889512684879872
</citizenWorkspaceAccountID>
    </cas:CaseInfo>
         </rem:xmlMessage>
      </rem:updateExternalCase>
   </soapenv:Body>
</soapenv:Envelope>
```

Motivations

A description of the Universal Access motivations implementation and how to implement a motivation.

Motivations Overview

Motivations allow customers to define their own processes and make them available from the citizen portal, for example, Apply For Healthcare. A motivation consists of:

- An IEG script used to collect data from clients.
- A datastore schema used to define the structure of the data collected in the IEG script.
- A display ruleset defines how the motivation results page will appear.
- A data ruleset (optional) Provide for transferring data store entities into rule objects in a rule set that is separate from the display rule set.

Running a motivation ends in a configurable results page. The script is used to define a set of questions that is displayed to a citizen when they initiate a motivation. The datastore stores the answers provided by a citizen in the script. The display ruleset and data ruleset are used to output the results on the results page.

Rule Sets

Motivations define a display rule set and, optionally, a data rule set. The display rule set drives the display of the results page with different page elements on the results page relating to the contents of the results datastore. Whilst an abstract ruleset(MotivationRuleSet) is provided which customers may use for reference and/or extend, the contract for the results page is only driven by the results schema, ie whether a customer chooses to refer to/extend the out of the box abstract ruleset or not the output of the customers' rules must conform to the results schema.

Data Rule Sets

Data rule sets are used to meet common requirements for converting data captured in the IEG data store into rule objects. The Motivations runtime can store the converted data store entities either in the Display rule set or a data rule set. In some situations it is more practical to add the converted data into the data rule set. This is because it is best practice to have a separate eligibility rule set for each program. The display rule set depends on the eligibility rule sets but it is inadvisable to have a circular dependency from the eligibility rule sets back on the display rule set. For this reason the data store entities can be converted into rule objects in a separate data rule set and the eligibility rule sets can depend on that rule set without any circular dependencies arising.

Results Datastore Population

Population of the results datastore is based on mapping the output from the rules to the results datastore, hence the requirement that the requirement that the output of the rules be tailored to the results datastore schema. The actual mapping is for the most part automatic but customers can influence this mapping via annotations. The Motivation_Display_Element annotation's resultSchemaElement allows a RuleObject outputted by the rules to be mapped to an element in the result schema where the name of the RuleClass and the name of the logically equivalent element in the result schema are different. If the RuleClass name and the the name of the logically equivalent element in the result schema are the same, mapping is automatic and no annotation is required. Irrespective of the mapping being driven by an annotation or not, the mapping is driven by the contents of the result schema..

Mapping Rule Objects

When a RuleObject (ie an attribute of a RuleObject which is itself another RuleObject) is added to the datastore, it is added as a new datastore entity as a child of the datastore entity which was added for the parent RuleObject. This only happens in the instance that the RuleObject's rule object name exists in the schema as a child element of an element, where the parent element's name matches the rule object name of the parent RuleObject. The matching here is case sensitive, person in the RuleObject and Person in the schema will not be considered a match. A RuleObject's rule object name is the resultSchema attribute of the Motivation_Display_Element, if an annotation exists. If no annotation exists, a RuleObject's rule object name is the name of the RuleClass. This is important because this is what is used for comparison with and mapping to the datastore, where appropriate based on what the schema allows.

When a simple attribute (an attribute of a RuleObject which is not itself a RuleObject, example a String or codetable value) is added to the datastore, it is added as an attribute of the datastore entity which was added for the RuleObject the attribute is on. This only happens in the instance that an attribute with the same name exists on the element in the schema which corresponds to the owning RuleObject's datastore entityType. The comparison of attribute names is case sensitive, dateOfBirth in the RuleObject and dateOFBIRTH in the schema will not be considered a match.

Sample Mapping From Rules Output To Datastore

Presuming a MotivationType record exists which specifies the following results schema, note that not all elements here are in the abstract rule set or required in the results datastore, they are just present for demonstrations purposes. These demonstrate how a customer would go about getting custom data from their ruleset populated in the result datastore.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:d="http://www.curamsoftware.com/BaseDomains" elementFormDefault="qualified">
    <xsd:import namespace="http://www.curamsoftware.com/BaseDomains">
    <xsd:include schemaLocation="IEGDomains">
    <xsd:include schemaLocation="MotivationResultDomains">
    <xsd:element name="Eligibility">
        <xsd:complexType>
            <xsd:sequence minOccurs="0">
                <xsd:element ref="Context" minOccurs="0" maxOccurs="1">
                <xsd:element ref="Results" minOccurs="0" maxOccurs="1">
                <xsd:element ref="ElementNameFromAnnotation" minOccurs="0" maxOccurs="1">
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Context">
        <xsd:complexType>
            <xsd:sequence minOccurs="0">
                <xsd:element ref="Person" minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="Summary" minOccurs="0" maxOccurs="1">
            </xsd:sequence>
            <xsd:attribute name="extratAttributeNotFromAbstractRuleSet" type="IEG_INT64">
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="ElementNameFromAnnotation">
        <xsd:complexType>
            <xsd:attribute name="attributeFromAnnotation" type="IEG_INT64">
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Person">
        <xsd:complexType>
            <xsd:attribute name="personID" type="IEG_INT64">
            <xsd:attribute name="firstName" type="IEG_STRING">
            <xsd:attribute name="lastName" type="IEG_STRING">
            <xsd:attribute name="dateOfBirth" type="IEG_DATE">
            <xsd:attribute name="status" type="CW_MOTIVATION_RESULTS_MEMBER_STATUS">
            <xsd:attribute name="gender" type="IEG_GENDER">
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Summary">
        <xsd:complexType>
            <xsd:attribute name="isRichText" type="IEG_BOOLEAN">
            <xsd:attribute name="summaryText" type="IEG_STRING">
            <xsd:attribute name="title" type="IEG_STRING">
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="Results">
        <xsd:complexType>
            <xsd:sequence minOccurs="0">
                <xsd:element ref="Category" minOccurs="0" maxOccurs="unbounded">
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="Category">
        <xsd:complexType>
            <xsd:sequence minOccurs="0">
                <xsd:element ref="Result" minOccurs="0" maxOccurs="unbounded">
            </xsd:sequence>
            <xsd:attribute name="categoryID" type="IEG_STRING">
            <xsd:attribute name="type" type="IEG_STRING">
            <xsd:attribute name="isPrimary" type="IEG_BOOLEAN">
            <xsd:attribute name="order" type="IEG_INT16">
            <xsd:attribute name="help" type="IEG_STRING">
            <xsd:attribute name="status" type="IEG_STRING">
            <xsd:attribute name="extratAttributeNotFromAbstractRuleSet" type="IEG_INT64">
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Result">
        <xsd:complexType>
            <xsd:sequence minOccurs="0">
                <xsd:element ref="Person" minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="Benefit" minOccurs="0" maxOccurs="unbounded">
```

Conceptually, the above schema only allows the datastore to be populated by the rules as follows. Anything else output by the rules will be ignored.

- Benefit.benefitType
- Benefit.benefitValue
- Benefit.explanation
- Result.Person
- Result.Benefit
- Result.resultID
- Result.type
- Result.resultDescription
- Result.status
- Category.Result
- Category.categoryID
- Category.type
- Category.isPrimary
- Category.order
- Category.help
- Category.status
- Category.extratAttributeNotFromAbstractRuleSet
- Context.Person
- Context.Summary
- Context.extratAttributeNotFromAbstractRuleSet
- Results.Category
- ElementNameFromAnnotation.attributeFromAnnotation
- Eligibility.Context
- Eligibility.Results
- Eligibility.ElementNameFromAnnotation
- Person.personID
- Person.firstName
- Person.lastName
- Person.dateOfBirth
- Person.status
- Person.gender
- Summary.isRichText
- Summary.summaryText
- Summary.title

Sample Rules: Processing Rule Objects

```
<Class name="Eligibility" extends="AbstractEligibility"
extendsRuleSet="MotivationRuleSet"
xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">
    <Attribute name="context">
       <type>
        <ruleclass name="AbstractContext" ruleset="MotivationRuleSet">
       </type>
         <derivation>
          <create ruleclass="Context">
         </derivation>
    </Attribute>
    <Attribute name="results">
       <type>
        <ruleclass name="AbstractResults" ruleset="MotivationRuleSet">
       </type>
         <derivation>
          <create ruleclass="Results">
         </derivation>
        </Attribute>
    <Attribute name="annotatedAttributeElementWillBeAnnotated">
       <type>
        <ruleclass name="AnnotatedElement">
       </type>
        <derivation>
          <create ruleclass="AnnotatedElement">
        </derivation>
    </Attribute>
</Class>
```

Figure 10. Processing Rule Objects Sample

Taking the RuleClass above, where Eligibility is the first RuleClass in the ruleset (this must always be the case), the motivation processing will add a datastore entity named Eligibility, and for each of the Context, Results and AnnotatedElement attributes process their attributes (and their attributes' attributes etc), adding new datastore entities and attributes to existing entities as appropriate relative to the schema.

Sample Rules: Complex Attributes (Single Rule Object)

```
<Class name="Eligibility" extends="AbstractEligibility"
extendsRuleSet="MotivationRuleSet"
xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">
        <Attribute name="context">
            <type>
                <ruleclass name="AbstractContext" ruleset="MotivationRuleSet">
            </type>
            <derivation>
                <create ruleclass="Context">
            </derivation>
        </Attribute>
        ... Other attributes ..
    </Class>

   <Class name="Context" extends="AbstractContext" extendsRuleSet="MotivationRuleSet"
        xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">
        ... Attributes ...

    </Class>
```

Figure 11. Complex Attributes

Context has not been annotated so the name used for the corresponding datastore entity will be the RuleClass name, ie Context. The processing will check the schema to see if Eligibility.Context is allowed (ie a combination of the parent RuleObject's name and this RuleObject's name), the schema does allow this so a Context entity will be added and appended to the Eligibility entity. For attributes that refer to a RuleObject, the name of the attribute on the parent RuleClass is not important (ie in the Eligibility RuleClass above, the name of context attribute is ignored). The matching is based on the name of the parent RuleObject and the name of the RuleObject itself, not the name of the attribute that refers to the RuleObject on the parent RuleObject.

Sample Rules: Complex Attributes (Single Rule Object, Annotated)

```
<Class name="Eligibility" extends="AbstractEligibility"
extendsRuleSet="MotivationRuleSet"
xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">
         .. Other attributes ..
        <Attribute name="annotatedAttributeElementWillBeAnnotated">
            <type>
                <ruleclass name="AnnotatedElement">
            </type>
            <derivation>
                <create ruleclass="AnnotatedElement">
            </derivation>
        </Attribute>
    </Class>

   <Class name="AnnotatedElement"
        xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">
        <Annotations>
            <Motivation_Display_Element resultSchemaElement="ElementNameFromAnnotation">
        </Annotations>

        .. Attributes ..

    </Class>
```

Figure 12. Complex Attributes (Single Rule Object, Annotated)

AnnotatedElement has been annotated so the name used for the corresponding datastore entity will be the resultSchemaElement attribute of the annotation, ie ElementNameFromAnnotation. The processing will check the schema to see if Eligibility.ElementNameFromAnnotation is allowed (ie a combination of the parent RuleObject's name and this RuleObject's name), the schema does allow this so an ElementNameFromAnnotation entity will be added and appended to the Eligibility entity. For attributes that refer to a RuleObject, the name of the attribute on the parent RuleClass is not important (ie in the Eligibility RuleClass above, the name of annotatedAttributeElementWillBeAnnotated attribute is ignored). The matching is based on the name of the parent RuleObject and the name of the RuleObject itself, not the name of the attribute that refers to the RuleObject on the parent RuleObject. Note that if the annotation was not present, the name that would be used for the potential datastore entity would be AnnotatedElement. The schema does not allow Eligibility.AnnotatedElement and so this entity would not be created. This demonstrates that a RuleClass which is intended to be present in the results datastore must be annotated to match the expected elements in the schema.

Sample Rules: Complex Attributes (List Of Rule Objects)

```
<Class name="Context" extends="AbstractContext" extendsRuleSet="MotivationRuleSet"
      xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">
      <Attribute name="householdMembers">
         <type>
              <javaclass name="List">
                  <ruleclass name="AbstractPerson" ruleset="MotivationRuleSet">
              </javaclass>
          </type>
          <derivation>
              <readall ruleclass="Person">
          </derivation>
      </Attribute>
      .. Other attributes ..
  </Class>

 <Class name="Person" extends="AbstractPerson" extendsRuleSet="MotivationRuleSet"
 xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">
   .. Attributes ..
  </Class>
```

Figure 13. Complex Attributes (List Of Rule Objects)

The attribute householdMembers above is a complex attribute, returning a List of RuleObjects. Each of the RuleObjects in the list will result in a new datastore entity being created and appended to the Context datastore entity, provided the name of these RuleObjects is appropriate for the schema. In this case, the processing will check the schema to see if Context.Person is allowed (ie a combination of the parent RuleObject's name and this RuleObject's name), the schema does allow this so a number of Person entities will be added and appended to the Context entity. For attributes that refer to a List of RuleObjects, the name of the attribute on the parent RuleClass is not important (ie in the Context RuleClass above, the name of householdMembers attribute is ignored). The matching is based on the name of the parent RuleObject and the name of each RuleObject itself, not the name of the attribute that refers to each RuleObject on the parent RuleObject.

Sample Rules: Simple Attributes

```
<Class name="Person" extends="AbstractPerson" extendsRuleSet="MotivationRuleSet"
        xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/CreoleRulesSchema.xsd">

        <Attribute name="personID">
            <type>
                <javaclass name="Long">
            </type>
            <derivation>
                <specified>
            </derivation>
        </Attribute>

        <Attribute name="medicaidCategory">
            <type>
                <codetableentry table="MotivationTestCategory">
            </type>
            <derivation>
                <specified>
            </derivation>
        </Attribute>

    </Class>
```

Figure 14. Simple Attributes

The attributes above, personID and medicaidCategory are simple attributes, ie they are not RuleObjects. Rather than being added as child entities in the datastore, they will be added as attributes of the datastore entity created for their parent RuleObject, provided the names of the attributes are appropriate for the schema. Unlike attributes which are RuleObjects where the name of the attribute in the parent RuleObject is not important, for simple attributes the attribute name IS important. The processing will check whether Person.personID and Person.medicaidCategory are appropriate for the schema. Person.personID is appropriate so the personID will be added as an attribute of the Person entity. Person.medicaidCategory is not contained in the schema and this will not be added to the Person entity in the datastore.

Fully Customizable Universal Access Artifacts

A description of the artifacts that are fully customizable in Universal Access and how to customize these artifacts.

Customizable Universal Access Page Content

Previous sections in this guide such as Customizing Existing Pages have discussed how to customize UIM pages that are used by Citizen Account. Some parts of the public application that are accessed by the publiccitizen user or generated users accounts use non-UIM pages. These pages are referred to as “Page Player Pages”. Customers cannot add new Page Player pages and are restricted from overriding the content of page player pages. It is possible, however to customize the content of these pages according to certain constraints.

Each page has a corresponding property file and images used in rendering these pages that are also stored in the app resource store. To navigate to the application resource store you must log in to the administration application, go to Universal Access, select Application Resource and filter your search here. Text, online help, and images for page content are all customizable and localizable.

Text and Online Help

Initial data for text and online help used in Universal Access pages are found in:

- *CURAM_DIR\EJBServer\components\CitizenWorkspace\Data_Manager\Initial_Data\blob\prop directory*

Universal Access makes use of the Application Resource Store mechanism to configure online help, images and page text for our pages. Taking the example of help text – this can be associated with any page in the Universal Access application. The help is displayed in a hidden panel at the top of the page which the user can access using the 'Help' link.

Every Universal Access page has a corresponding application resource of type 'property' shipped with it. To change the online help for one of the Universal Access pages, the developer needs to know the name of the page and the corresponding properties file. For example, the 'ScreeningOptionalLogin' page (the Getting Started page that is displayed after selecting 'Am I Eligible' on the Citizen Portal Home page). The corresponding properties file can be found at:

- *CURAM_DIR\EJBServer\components\CitizenWorkspace\Data_Manager\Initial_Data\blob\prop\ScreeningOptionalLogin.properties*

This is referenced from the DMX file:

- *CURAM_DIR\EJBServer\components\CitizenWorkspace\Data_Manager\Initial_Data\APPRESOURCE_PROP.dmx*

As a change is being made to initial DMX data, the procedure to follow is the same as the recommended procedure for changing any DMX data as outlined in the `Cúram Server Developer's Guide`.

Simply edit your version of the ScreeningOptionalLogin.properties file and change the property text as required. All text controlled by page Player XML properties files can be altered in the same manner.

Images

Initial data for images used on the Universal Access pages are found in:

- *CURAM_DIR\EJBServer\components\CitizenWorkspace\Data_Manager\Initial_Data\blob\img*

The process for replacing icons/images is the same as that used to replace text. For example, take the 'ScreeningOptionalLogin' page. The page XML source file is located at:

- *Data_Manager\Initial_Data\blob\xml\ScreeningOptionalLogin*

Note that the icon associated with the page header is the "title_getting_started" icon. This file is located at:

- *Data_Manager\Initial_Data\blob\img*

To replace this image with another, follow the same process indicated for replacing page text/help text above.

Translation

It is possible to use separate properties files to provide translations of Page Content to different languages. The following example shows how to add a new translation for the ScreeningOptionalLogin page. Broadly speaking, this example follows the guidelines for adding new entries to DMX files as described in the *IBM Cúram Server Developer's Guide*.

To create a French translation of the ScreeningOptionalLogin page, create a new DMX file,

- *CURAM_DIR\EJBServer\components\custom\Data_Manager\Initial_Data\APPRESOURCE_PROP.dmx.*

Add a row to this file which references a new file blob\prop\ScreeningOptionalLogin_fr.properties. The resource name needs to be the same as the resource name for the English version of the properties, i.e. ScreeningOptionalLogin. However, the 'localeIdentifier' column will contain <value>fr</value>.

Add a new entry to project\properties\datamanager_config.xml which references:

- *CURAM_DIR\EJBServer\components\custom\Data_Manager\Initial_Data\APPRESOURCE_PROP.dmx*

Create the file CURAM_DIR\EJBServer\components\custom\Data_Manager\Initial_Data \blob\prop\ScreeningOptionalLogin_fr.properties and enter French translations for all relevant property values.

For information about adding new languages to citizen account pages, see "Customizing the citizen account".

Universal Access Page Player Look and Feel

The look and feel of the Universal Access can be changed (to a certain extent) through changing/customizing its appearance properties and style sheet. The general appearance properties are initialized from the file:

- *CitizenWorkspace\Data_Manager\Initial_Data\blob\css\cp-config.properties*

It is referred to by the Application Resource name cp-config-properties.

The main style sheet is initialized from:

- *CitizenWorkspace\Data_Manager\Initial_Data\blob\css\cp-css-template.css*

It is referred to by the Application Resource name cp-css-template.

The banner is similarly initialized from:

- *CitizenWorkspace\Data_Manager\Initial_Data\blob\css\banner-css-template.css*

It is referred to by the Application Resource name banner-css-template.

Note the use of properties in the.css files such as 'banner.icon'. When Universal Access loads the style sheet template, it substitutes these properties from cp-config.properties into the template to create the actual style sheet, so many aspects of the page player appearance can be changed simply by changing this properties file without any need to modify the.css files. As with the previous examples in this section the css-templates and associated properties can be changed through taking a copy of the application resource DMX data into the custom component.

For information regarding customizing the look and feel of citizen account pages, see "Customizing the citizen account".

General Universal Access Settings

The file:

- *CitizenWorkspace\Data_Manager\Initial_Data\blob\prop\CPPagePlayer*.properties*

and its translated equivalents like CPPagePlayer_es.properties, control general purpose text and images associated with the Universal Access application. For example, text for 'Next' and 'Back' buttons, text on page banners, etc. This resource is registered under the resource name 'CPPagePlayer' and can be changed in the same manner described by the sections above concerning Content/Help text.

Customizable Universal Access Public APIs

The CitizenWorkspace and WorkspaceServices components contain APIs. The javadoc for these APIs can be located in the doc sub-directory of each of

- *<CURAM_DIR>\EJBServer\components\CitizenWorkspace\doc* and
- *<CURAM_DIR>\EJBServer\components\WorkspaceServices\doc* respectively.

A limited number of these APIs are customizable by Event and Strategy patterns as described in the *IBM Cúram Development Compliancy Guide*.

Extendable Code Tables

Customers are advised to refer to the *IBM Cúram Development Compliancy Guide* for a list of restricted code tables.

Universal Access Artifacts with Limited Scope for Customization

A description of the included Universal Access artifacts that have restrictions on their use. Customers that are looking to change these artifacts should consider alternatives or request an enhancement to Universal Access.

Model

Customers are not supported in making changes to any part of the Universal Access model. Changes in the model such as changing the data types of domains are likely to cause failure of the Universal Access system and upgrade issues. This applies to the model files in the following packages:

- WorkspaceServices
- CitizenWorkspace
- CitizenWorkspaceAdmin

Universal Access Page Player XML

Customers are not supported in making changes to the Page Player XML files.

JSP and JSPX pages

Customers are not supported in making changes to any default JSP or JSPX files.

Javascript files

Customers are not supported in making changes to any default JavaScript files in the following components:

- WorkspaceServices
- CitizenWorkspace
- CitizenWorkspaceAdmin

Renderer configuration

Customers are not supported in making changes to any of the renderer configuration files.

This applies to the XML files in the following locations:

- webclient\components\WorkspaceServices\Configuration
- webclient\components\CitizenWorkspace\Configuration
- webclient\components\CitizenWorkspaceAdmin\Configuration

Client-side Java artifacts

Universal Access delivers all of its client-side Java artifacts in CitizenWorkspace_source.jar. This JAR file contains all of the classes required for UA renderers and servlets. Customers are not supported to attempt to extend, modify, or replace any of the delivered classes.

Code Tables

Customers are advised to refer to the *IBM Cúram Development Compliancy Guide* for a list of restricted code tables.

Page `playerID` Uniqueness in a WebSphere Clustered Environment

A page `playerID` that gets generated upon the start of an intake application is stored as a name-value pair within a servlet context attribute. No two `playerID`s are the same, ensuring thread safe functioning on a clustered environment. These IDs are generated on the server side through the `curam.core.fact.UniqueIDFactory` application programming interface (API), which is through a key server .dmx file.

Notices

Privacy Policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies or other similar technologies that collect each user's name, user name, password, and/or other personally identifiable information for purposes of session management, authentication, enhanced user usability, single sign-on configuration and/or other usage tracking and/or functional purposes. These cookies or other similar technologies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

Printed in USA