IBM Cúram Social Program Management

Cúram Universal Access Customization Guide

Version 6.0.4

Note

Before using this information and the product it supports, read the information in Notices at the back of this guide.

This edition applies to version 6.0.4 of IBM Cúram Social Program Management and all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Chapter 1 Introduction 1
1.1 Purpose 1
1.2 Audience 1
1.3 Scope 1
1.4 What You Need to Know 2
1.5 Chapters in this Guide 2

Chapter 2 The Black Box Engineering Philosophy 4
2.1 Introduction 4
2.2 It Saves Time and Money 4
2.3 It Makes For Easier Upgrades 4
2.4 It Is Still Configurable and Customizable 5

Chapter 3 Securing Universal Access 6
3.1 Introduction 6
3.1.1 Background 6
3.2 The Universal Access Security Model 7
3.2.1 The Public Citizen Account 7
3.2.2 Anonymous Accounts 7
3.2.3 Registered Accounts 7
3.2.4 Linked Accounts 8
3.2.5 Authorization Roles and Groups 8
3.3 Deployment Considerations 9
3.4 Managing Usernames and Passwords 10
3.4.1 Account Management 10
3.5 Data Caching 11
3.5.1 Browser Caching 12
3.6 External Security Authentication 12
3.6.1 Analysis 12
3.6.2 Example 12
3.6.3 Configuration Tasks 13
3.6.4 Configure the Application Server to use LDAP for Authentication 13
3.6.5 Deploy Cúram Universal Access in Identity Only mode for Registered Users 14
3.6.6 Configure Cúram Universal Access so that Create Account Screens are not Displayed 15
3.6.7 Configure Cúram Universal Access so that users are directed to register with

an External System 16
3.6.8 Development Tasks 16

Chapter 4 Customizing Universal Access Triage 19
4.1 Introduction 19
4.2 Available triage events 19
4.2.1 Standard persistence events 19
4.2.2 Triage Referral Event 19

Chapter 5 Customizing Universal Access Screening 21
5.1 Introduction 21
5.2 How to Track the Volume, Quality, and Results of Screenings 21

Chapter 6 Customizing Application Intake Processing 22
6.1 Introduction 22
6.2 How to Pre-populate the Intake Script 22
6.3 How to Add a Validation for Program Selection 22

Chapter 7 Customizing the Handling of Submitted Applications 24
7.1 Introduction 24
7.2 How to Customize the Generic PDF for Processed Applications 24
7.3 How to Use Events to Extend Intake Application Processing 25
7.4 How to Send Applications to Remote Systems for Processing 25
7.5 How to Customize the Process Intake Application Workflow 26

Chapter 8 Customizing Citizen Account 27
8.1 Introduction 27
8.2 Technical Overview 27
8.3 Security Considerations 28
8.3.1 Ensuring the currently logged in user is of the correct type 28
8.3.2 Ensuring the currently logged in user has access to the specific records they have requested. 29
8.4 How to Add a New Page to Citizen Account 29
8.4.1 Create a custom, external client component 30
8.4.2 Create a UIM page in the new component 30
8.4.3 Add a navigation entry for the new page 30
8.4.4 Create a Facade 30
8.5 How to Customize Universal Access Style Sheets in Citizen Account 31
8.6 Customizing Locale 31
8.7 Citizen Account Homepage 32
8.7.1 Customizing display text 32
8.7.2 Outreach Campaigns 32
8.7.3 My Messages 37
8.8 Customizing existing pages 47
8.9 My Payments Page Customization 47
8.10 My Applications Page Customization 47
8.11 Contact Information Page Customization 48

Chapter 9 Customizing Life Events 49
9.1 Purpose 49
9.2 Audience 49

9.3 Overview 49
9.4 Introduction to Life Events 49
9.5 How to Build a Life Event 51
9.5.1 Analysis 51
9.5.2 Building The Components of a Life Event 53
9.6 Life Events API Guide 74
9.6.1 Event APIs for Life Events 74

Chapter 10 Universal Access Web Services 75
10.1 Introduction 75
10.2 Web Services Security Considerations 76
10.3 Process Application Service 76
10.3.1 Receive Application 76
10.3.2 Receive Withdrawal Request 77
10.4 Update Application Service 79
10.4.1 Intake Program Application Update 79
10.4.2 Withdrawal Request Update 79
10.5 Life Event Service 80
10.6 Create Account Service 81
10.7 Link Service 82
10.8 Unlink Service 83
10.9 Citizen Message 84
10.10 Payment Service 85
10.11 Contact Service 87
10.12 Case Service 87

Chapter 11 Fully Customizable Universal Access Artifacts 89
11.1 Introduction 89
11.2 Customizable Universal Access Page Content 89
11.2.1 Text and Online Help 89
11.2.2 Images 90
11.2.3 Translation 90
11.2.4 Universal Access Page Player Look and Feel 91
11.2.5 General Universal Access Settings 92
11.3 Customizable Universal Access Public APIs 92
11.4 Extendable Code Tables 92

Chapter 12 Universal Access Artifacts with Limited Scope for Customization 93
12.1 Introduction 93
12.2 Model 93
12.3 Universal Access Page Player XML 93
12.4 JSP and JSPX pages 93
12.5 Javascript files 94
12.6 Renderer configuration 94
12.7 Client-side Java artifacts 94
12.8 Code Tables 94

Appendix A Sample SOAP Requests 95
A.1 Intake Program Application Update 95
A.2 Withdrawal Request Update 95

A.3 Create Account 96
A.4 Account Link 97
A.5 Account UnLink 97
A.6 Citizen Message 98
A.7 Payment (Simple) 98
A.8 Payment (Batched) 99
A.9 Contact 100
A.10 Cases 100
Notices 102

Chapter 1

Introduction

1.1 Purpose

The purpose of this guide is to describe options and provide instructions for customizing IBM Cúram Universal Access™ (UA) components. Customization can be distinguished from configuration in that customization allows developers to modify, extend, or replace source code to suit customer requirements. The components that make up Universal Access that we are concered with are CitizenWorkspace, CitizenWorkspaceAdmin and WorkspaceServices. The major customizable functional areas discussed in this document are Triage, Screening, Intake, Security, Citizen Account and Life Events. Please read the `Cúram Universal Access Guide` to become more familiar with these concepts.

Please note, the IBM Cúram Universal Access product is implemented by a collection of components listed above. These components are collectively known as the Citizen Workspace.

Also note that throughout this guide Universal Access is referred to as 'UA', the corollary of this is that the acronym is avoided when making reference to cascading style sheets.

1.2 Audience

This guide is intended for developers responsible for customizing Universal Access components.

1.3 Scope

This guide covers the customization of the CitizenWorkspace, CitizenWorkspaceAdmin and WorkspaceServices components. Customers licensed for the Cúram Enterprise Framework™ but not Universal Access need only

concern themselves with workspace services related customization points.

Note

This guide does not cover the configuration options available for the UA component. Configuration allows administrators to determine the information that is displayed on application pages. The Cúram Universal Access Configuration Guide describes this configuration in some detail.

1.4 What You Need to Know

Before reading this guide you should be familiar with the contents of the following developer guides. These developer guides explain the basics of how to configure the behavior of the UA component and how to perform customization within the product set in general, with an emphasis on events and workflow.

- Using the Data Mapping Engine
- Cúram Universal Access Guide
- The Cúram Development Compliancy Guide
- Cúram Server Developer's Guide
- Persistence Cookbook
- Cúram Workflow Management System Guide

1.5 Chapters in this Guide

The Black Box Engineering Philosophy

This chapter provides a brief discussion of the UA commitment to the black box engineering philosophy.

Customizing Triage

This chapter discusses the customization points that exist around the Triage process

Customizing Universal Access Screening

This chapter discusses the customization points that exist around the screening process.

Customizing Application Intake Processing

This chapter discusses the customization points that exist around the Intake process prior to submission.

Customizing the Handling of Submitted Applications

This chapter discusses the customization points that exist around the intake process post-submission.

Customizing Citizen Account

This chapter discusses the customization points that exist around the Citizen Account.

Customizing Life Events

This chapter discusses the customization points that exist around Life Events.

Universal Access Web Services

This chapter describes the UA web services and how to develop peer code to communicate with those web services.

Fully Customizable Universal Access Artifacts

This chapter describes the artifacts delivered that are fully customizable.

Universal Access Artifacts with Limited Scope for Customization

This chapter indicates the artifacts delivered that have restrictions on their use.

Chapter 2

The Black Box Engineering Philosophy

2.1 Introduction

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 guide, 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.

2.2 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.

2.3 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 up-

grade 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.

2.4 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 document will describes these options.

Chapter 3

Securing Universal Access

3.1 Introduction

The purpose of this chapter is to:

- Provide readers with an overview of the Universal Access Security Model
- Provide developers with an understanding of to how customize Universal Access securely

3.1.1 Background

Universal Access is designed to give citizens access to their most sensitive personal data via the 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: The remainder of this chapter discusses the Universal Access Security Model and provides the reader with information on how to add customizations that work with rather than against this Security Model.

- Ensure that the project team are familiar with the the principles of secure application development, and common vulnerabilities such as the OWASP Top Ten [http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project]
- Develop a Threat Model [http://www.owasp.org/index.php/Threat_Risk_Modeling] 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

3.2 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. This section introduces the different account types and explains why they are needed. The user types can be summarized as:

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

3.2.1 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. .

3.2.2 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.

3.2.3 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.

3.2.4 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 into 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. Multi-Factor authentication is discussed later in this chapter .

3.2.5 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, addditional permissions will be required.

This is explained in the chapter regarding the customisation of 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. This is discussed in the chapter that discusses citizen account customisation.

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. Please see the Citizen Account - Security Considerations section for more information.

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. Please refer to the Citizen Account - Security Considerations section for more information.

3.3 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 caseworkers 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 regarding securing citizen account,

please see the Citizen Account - Security Considerations section.

3.4 Managing Usernames and Passwords

There is a range of ways to customise 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.

3.4.1 Account Management

This section describes the way you can customise 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:

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.

Table 3.1 Account configurations

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

Event Interface	Description
CitizenWorkspaceCreateAccountEvents	Events raised around account creation. Please see the related javadoc in the WorkspaceServices compon-

Event Interface	Description
	ent for more information.
CitizenWorkspacePassword-ChangedEvent	Event raised when a user is changing their password. Please see the related javadoc in the WorkspaceServices component for more information.
CitizenWorksapceAccoun-tAssociations	Events raised when a user is linked or unlinked from an associated Person Participant. Please see the related javadoc in the WorkspaceServices component for more information.

Table 3.2 Account events

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 caseworkers 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.

3.5 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.

3.5.1 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.

3.6 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.

3.6.1 Analysis

This section discusses, by way of example, the analysis 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?

3.6.2 Example

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 into 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 into 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.

3.6.3 Configuration Tasks

Taking the example from the Analysis section 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.

The following sections will elaborate upon the above configuration tasks.

3.6.4 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.

3.6.5 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
```

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. 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.
7. Log in as sysadmin.
8. Select Application Data -> Property Administration.
9. Select Category ”Infrastructure – Security parameters”.
10. Set curam.custom.externalaccess.implementation to 'curam.citizenworkspace.security.impl.CitizenWorkspacePublicAccessSecurity'.
11. Publish the property changes.

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

3.6.6 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.

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

Out of the Box Universal Access invites users to login 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 Section 11.2 *Customizable Universal Access Page Content* in this Guide.

3.6.8 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 into 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.SecurityStrategy class and overriding the authenticate() method. The code below outlines how to use the SecurityStrategy and other security APIs to meet the requirements described above:

```
public class CustomSecurityStrategy extends SecurityStrategy {
  @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 SecurityStrategy. In the case of a registered user the Security 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 encrypted using a one-way hash. This ensures that it can be safely transmitted from the Client Side of the Cúram application to the Server Side of the application.

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>
```

Chapter 4

Customizing Universal Access Triage

4.1 Introduction

This chapter details customization points around the triage process. For information on configuring & administring triage please refer to the `Cúram Universal Access Guide`.

4.2 Available triage events

There are a number of events which are fired during the triage process. These events can be broken into two categories, persistance events and custom events. The persistance events are standard data access events fired by the persistance 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.

4.2.1 Standard persistence events

On execution of the triage ruleset, the results of the session are persisted to the `TriageResult` entity. This will trigger the invocation of the pre and post insert events. For details on how to make use of the `PersistenceEvent` API please refer to the `Persistence Cookbook` (Chapter 6).

4.2.2 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 further details, refer to the API JavaDoc for ReferralEvent. This can be found in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/`

doc

Chapter 5

Customizing Universal Access Screening

5.1 Introduction

This chapter details customization points around the screening process. For information on setting up and configuring screening please refer to Chapter 5 of the Cúram Universal Access Configuration Guide.

5.2 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

Chapter 6

Customizing Application Intake Processing

6.1 Introduction

There are a number of customization points available to customers that can be built upon to customize the application intake process. This chapter covers the process of intake up to the point of submitting an intake application. For more information on configuring an intake application please see the Cúram Universal Access Guide.

6.2 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 Datastore 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

6.3 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

Chapter 7

Customizing the Handling of Submitted Applications

7.1 Introduction

There are a number of customization points available during the process of application intake. This chapter covers those that occur after an intake application has been submitted. For more information on configuring an intake application please see the Cúram Universal Access Guide.

7.2 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.

7.3 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 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 for `IntakeProgramApplication.IntakeProgramApplicationEvents` . This can be found in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc` .

7.4 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 Datastore 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 .

7.5 How to Customize the Process Intake Application Workflow

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. This workflow creates an integrated case with a status of 'application'. Core case and common evidence data are also mapped to this integrated case. This process can be intercepted using the postMapData-ToCuram event described above.

Chapter 8

Customizing Citizen Account

8.1 Introduction

The Citizen Account is a facility within Universal Access that allows a linked UA user to login to a secure area where they can screen and apply for programs. The citizen can also 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 may be relevant to them. Citizen Account also provides a framework for customers to build their own citizen account pages or override the existing pages.

Please refer to the Cúram Universal Access Guide for a full description of all the functionality offered out of the box, and more information on linked UA users.

8.2 Technical Overview

Unlike the rest of the UA application, the citizen account framework is defined in UIM. This means that customers can override existing pages, add their own, and customize the navigation of the framework as they can customize the caseworker application.

Citizen account is built on top of the user interface infrastructure. It uses only a sub set of the user interface and navigation components offered by the infrastructure, in order to achieve a simple, usable application that citizens can understand and use without any specific training.

Linked UA users will perform UA actions of triage, screening and online application via their account. Accordingly, citizen account includes UIM pages that are a view onto the triage, screening and online application functionality. These pages are not configurable or customizable: the functionality that they offer is configurable via administration as specified in the documentation for these areas. The UIM pages related to triage, screening and

online application should not be modified or overridden.

8.3 Security Considerations

Exposing sensitive data to citizens over the web is inherently dangerous and security must be a primary concern when developing citizen account customizations. Please see the chapter Securing Universal Access for more information. We strongly recommend that all public facing applications undergo rigorous security analysis and testing before being deployed. We also recommend that you contact support to discuss unusual customizations that may have specific security issues.

Permission to invoke the server facade methods that serve data to citizen account pages is managed by the standard authorization model. Please see the Cúram Server Developer's Guide for more information. In addition to the standard authorization checks, each facade method that is invoked by a citizen account page must perform the following security checks in order to ensure the user 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, i.e. validate any page parameters passed in to ensure that the records requested are related to the currently logged in user in some way.

8.3.1 Ensuring the currently logged in user is of the correct type

The curam.citizenaccount.security.impl.CitizenAccountSecurity API offers a method performDefaultSecurityChecks that will ensure that the user is of the correct type. This method will check the user type, and if not acceptable, will write a message to the logs and fail the transaction. *This should be called in the first line of every custom facade method, before any processing or further validation has taken 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();
  }
```

8.3.2 Ensuring the currently logged in user has access to the specific records they have requested.

A malicious user logged in to a valid linked UA account could send requests to the system requesting data related to other users. In order to prevent this from happening, all page parameters must be validated to ensure that they are somehow traceable back to the currently logged in user. How this is determined is different for each type of record. For example, a Payment can be traced back to the Participant via the Case it was issued on.

The `curam.citizenaccount.security.impl.CitizenAccountSecurity` API offers methods to perform these checks for the types of records that are served to citizens by the OOTB pages. Please review the javadoc of this API for specific information. For custom pages that serve different kinds of data, additional checks must be implemented to validate the page parameters. These should be added to a custom security API and invoked by the façade methods in question. The methods should check to see if the record requested can be traced back to the currently logged in user, and if not, it should log the user name, method name and other data, and fail the transaction 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 should be logged, it is important to ensure that the exceptions thrown does not expose any information that may be useful to malicious users. A generic exception should be thrown, that does not contain any information relating to what went wrong. The `curam.citizenaccount.security.impl.CitizenAccountSecurity` API throws a generic message stating "You are not privileged to access this page."

8.4 How to Add a New Page to Citizen Account

This section explains the tasks that must be carried out in order to add a custom page to citizen account.

8.4.1 Create a custom, external client component

Artifacts that form part of public facing applications such as UA should be stored in separate components in order to avoid deploying internal pages intended for administrators or caseworkers into public facing applications. Accordingly, 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. Please refer to the instructions in section 'External Client Applications' of the Server Developer Guide on how to do this. This component should include the artifacts to be deployed to UA, and should not include any artifacts intended for use by administrators or caseworkers. This component must be added to the deployment packaging for the UA EAR file. The important point below is that the deployment_packaging.xml, shipped in the `<CURAM_DIR>/EJBServer/project/config/` folder will contain the minimum entries for the components that need to be built, namely CitizenAccount, CitizenWorkspace, IntelligentEvidenceGathering and CEFWidgets.

8.4.2 Create a UIM page in the new component

Develop the custom UIM page in the new client component. The user interface infrastructure offers a wide array of complex functionality. Bear in mind that the target audience of this page are citizens that will not be familiar with complex user interfaces, so 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.

8.4.3 Add a navigation entry for the new page

This is done in the standard way. Please refer to 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>
```

Example 8.1 sample custom navigation entry for custom citizen account page

8.4.4 Create a Facade

Develop a facade that the page can call. This facade will retrieve data based on either the currently logged in user, or page parameters that are passed in. Generally, citizen account pages read data related to the logged in user's

linked accounts. Specifically, if the logged in user is linked to a Cúram participant, i.e. a concernRoleID then data relating 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` API offers a convenience method that can be used to retrieve the linked identities of a currently logged in user including their linked ConcernRole if they have one. It is recommended that customers use this API to retrieve linked identities, as it has 'baked-in' security checks to ensure that the user in question is in fact a linked UA user.

Relevant authorization entries must be added in DMX in order to give the linked UA users permission to invoke the new facade method. Add an entry for the new method to the LINKEDCITIZENWORKSPACEGROUP. For example:

```
<row>
     <attribute name="groupName">
       <value>LINKEDCITIZENWORKSPACEGROUP</value>
     </attribute>
     <attribute name="sidName">
       <value>MyCustomFacadeName.myCustomFacadeMethodName</value>
     </attribute>
   </row>
```

8.5 How to Customize Universal Access Style Sheets in Citizen Account

Citizen account UA style sheets are customizable in the standard way for UIM pages. Please refer to the `Curam User Interface Developers Guide` for information. The citizen account style sheets are located in `webclient/components/CitizenAccount/css` .

8.6 Customizing Locale

It is important to note that the method for adding new locales to citizen account is the same as for standard UIM pages as opposed to the dynamic manner by which public UA pages can be internationalized. In order to add different locales to citizen account, the client project must be rebuilt to generate the JSPs in the new locale. Please refer to the Cúram Web Client Developer Guide for more information 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 should be routed to when they log in. The client infrastructure uses these configurations to determine where the user should be routed following a successful authentication. UA ships with 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 adding additional locales to UA, additional entries must be added to this codetable for each locale that is being added. All should contain the same description, which contains the value of the citizen account homepage. This is also 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. For more information on localisation see the chapter Fully Customizable Self Service Artifacts.

8.7 Citizen Account Homepage

The citizen account homepage offers a range of functionality to offer pertinent information to the citizen. Please refer to the Cúram Universal Access Guide . The various areas are configured in different ways, which are outlined in the Cúram Universal Access Configuration Guide.

When referring to customizing the Citizen Account homepage this guide is referring the constituent parts of this page, the Outreach (Did You Know?) and My Messages Panels.

8.7.1 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 located 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".

8.7.2 Outreach Campaigns

Outreach campaigns are designed to display targeted campaign messages to the citizen. These can include images and links to both UA pages and to external websites. Whether or not a specific campaign should be displayed to a particular citizen is determined by a 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. Please see the Cúram Universal Access Configuration Guide for more information.

How to configure a new Citizen Campaign

Please refer to the Universal Access Configuration Guide for information regarding configuring new Outreach campaigns in Administration.

Outreach Campaign Rulesets

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

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

CitizenCampaignAdmin rule class

This is a 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 are managed internally by the Outreach infrastructure.

AbstractCampaignAdviceItem rule class

This class extends AbstractAdviceItem (see Advisor documentation). This is the class 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. The citizenCampaignName specified in the rule set and the name of the Outreach campaign when it is created in Administration *must be identical* . Accordingly, when creating the new Outreach campaign 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 should return 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 (i.e. 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 allows administrators to configure the expiry of campaigns. However, it is also possible to determine the expiry

date time based on business logic or other rules if they so wish, 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. Please see the Advisor documentation and the following sample campaign rule set for more information.

```
<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 therefore Outreach support including images and links as part of an Advice Item / campaign. The image itself is uploaded when creating a new Outreach campaign. By default, if an image is specified when creating the campaign in Administration, it is displayed as part of the campaign

without a link. However, it is possible to specify a link within the rule set, and within that link to specify an image, referencing the image that has been 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 declaring this link within the custom implementation of AbstractCampaignAdviceItem, you specify the reference to the image configured in administration: Please refer to the Advisor documentation for further information on defining links and images.

```
<Attribute name="childCareOptionLinkWithImage">
   <type>
     <ruleclass name="ChildCareOptionLinkWithImage"/>
   </type>
   <derivation>
     <create ruleclass="ChildCareOptionLinkWithImage">
```

```
      <specify attribute="linkImage">
        <reference attribute="campaignImage"/>
      </specify>
    </create>
  </derivation>
</Attribute>
```

In order for the links related to image only campaigns to be persisted to the Advisor database tables (and therefore 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 references name of the rules object defined in the campaign rule set to represent this link.

Performance Considerations

Campaigns refer to CER rule objects in order to determine whether to display campaigns. Therefore, when the underlying data that these rule objects depend on changes, CER reassessment will be triggered. This will cause Advisor to re-calculate whether the campaign should be displayed or not. This can affect performance and needs to be considered. There are two different kinds of data changes involved:

Changes to the Participants' data

These kinds of changes affect a specific participant. Take for example, a campaign that references a citizen's address. Every time the user changes their address, this change would be propagated to the rule object representing that participants address. Because the campaign rule object is dependent on this, reassessment would be triggered. This means that every time the participant changes his address, the campaign rules will be executed to determine if it should still be displayed. Therefore, it is important to consider how often a piece of data may change, and for how many citizens, and whether referencing it in a campaign may 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 means that it will trigger reassessment for every citizen that has been assessed for eligibility for this campaign. For example, if the image associated with a campaign is changed, the system will re-execute the rules for each citizen that has been considered for this campaign. This could require a significant amount of processing that could have a performance impact on the system. Accordingly, we recommend that changes in campaign administration are performed when the system is not under a heavy load, or has been taken offline for maintenance.

8.7.3 My Messages

Please refer to the Cúram Universal Access Guide for an overview of the functionality offered by the messages panel and the specific messages offered out-of-the-box.

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

Configuring Citizen Messages

There are global configurations 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 regarding how to customize the various message types is provided in later sections of this document. Please refer to the Cúram Universal Access Configuration Guide for more information on how to change the global configurations and on delivering Citizen Messages using web services.

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

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

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

Table 8.1 Message Properties Files

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 have been persisted to the database via the `curam.citizenmessages.persistence.impl.ParticipantMessage` 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 on the fly by a listener that listens for the event that is raised when the citizen 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 may be of interest to the citizen, a message is persisted to the database. For example, when a meeting invitation is created, an event is fired. Our OOTB meeting messages functionality listens for this event, and if the meeting invitee is a participant with a linked UA account, a message is written to the ParticipantMessage table informing the citizen that they have been invited to the meeting.

One benefit of this approach is that there is very little processing performed when the citizen logs in to see this message: the message is simply read from the database and displayed, as opposed to calculation taking place that would determine if the message was required or not. However, the implementation needs to also handle any changes to the underlying data that may invalidate or change the message, and take appropriate action. For example, our meeting message functionality also listens for changes to meetings in order to ensure the meeting time, location etc are up to date, and to issue a new message to the citizen informing that the location or time has changed.

Dynamic Messages

These messages are generated when the citizen 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 citizen logs in, so if some underlying

data changes, the next time the citizen logs in, they will get the correct message.

A drawback to this approach is that significant processing may be required at runtime in order to generate the message. Care must be taken to ensure that this does not adversely affect the load time of the citizen account homepage.

Performance considerations must be evaluated against the effort involved to manage change to the data that the message is related to over time, and the requirements of the specific message type. For example, the OOTB verification message is dynamic, when a citizen logs in it checks to see if any outstanding verifications exist for that citizen. This is a relatively simple database read, whereas it would have been complicated to listen for various events in the Verification Engine and ensure an up to date message was stored in the database regarding the participants' outstanding verifications. On the other had, the meeting messages need to inform the citizen 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 `CitzenMessagesEvent.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 `CitzenMessagesEvent` . 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 homepage (there is also a public CitizenMessageController method that will return all messages for a citizen as a list, please see the javadoc)

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 of `curam.citizenmessages.persistence.impl.ParticipantMessage`, `curam.advisor.impl.Parameter` and

curam.advisor.impl.Link for more information.

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 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

The message types delivered OOTB are customizable in various ways that shall be described in this section. Please refer to the Cúram Universal Access Guide for a description of the various message types.

Payment Messages

This message type creates messages based on the payments issued, canceled etc 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.

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 event	Message Property
Payment canceled	Message.Cancelled.Payment
Payment reissued	Message.Reissue.Payment
Payment stopped (case suspended)	Message.Stopped.Payment
Payment / Case unsuspended	Message.Unsuspended.Payment

Table 8.2 Payment messages and related properties

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 overridden from property administration.

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

Table 8.3 Payment message expiry property

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 re-

gistered 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.

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.*

Table 8.4 Meeting messages

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 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.

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.

Table 8.5 Meeting message display date property

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.

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.

Table 8.6 Activity types for which to generate meeting messages

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 overridden from property administration.

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

Table 8.7 Application acknowledgment message expiry property

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 EJBServ-

er\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.

8.8 Customizing existing pages

The 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.

8.9 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.

8.10 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.

8.11 Contact Information Page Customization

This page displays caseworker 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.

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

Table 8.8 Contact Information Customization properties

Chapter 9

Customizing Life Events

9.1 Purpose

The purpose of this chapter is to:

- Describe what a Life Event is and why they are useful
- Describe how to develop Life Events

9.2 Audience

This chapter is intended for business analysts, software architects and developers. Many types of Life Events can be built entirely by analysts, some will require input from developers. The chapter 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.

9.3 Overview

This chapter aims to give the reader an overview of Life Events as well as a complete guide to developing Life Events.

Section 1 provides a brief introduction to Life Events

Section 2 describes the high level architecture of Life Events and details how to perform the analysis and development tasks in building a Life Event

Section 3 describes the Life Event Java API

9.4 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 has lost his job after the company he is working with shuts down. James logs into his Citizen Account and goes to the Life Event section. He chooses the "Lost my Job" Life Event. The system launches an IEG2 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 already known about James Smith. For example, the name and address of his former employer are displayed in the script. James confirms that indeed this is the employer that laid him off.

After completing the Life Event script, 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 may be relevant to James' situation, for example Unemployment Benefit

A couple of days after submitting the Life Event, James logs into 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 administering 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 he sent to the agency and also remind himself of the services 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 may 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 has applied for a program, they now know what has 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 using Cúram, then information from the Life Event can be fed automatically into these cases

9.5 How to Build a Life Event

9.5.1 Analysis

This section describes how to undertake the analysis needed to design a Life Event. This section is concerned with Life Events for Universal Access users. 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 chapter. Java coding skills are not a pre-requisite 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 section 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 into 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 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 section 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 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, a temporal evidence or participant-related data store in Cúram. These considerations should 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 IEG2 script?
3. What Temporal Evidence data is read by the Life Event?
4. What Temporal 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 will be 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 dealing with Non-Evidence Entities presents the greatest challenge. Any Life Event that updates non-evidence entities will require developers with Java skills.

9.5.2 Building The Components of a Life Event

Overview

This section describes how to build the component parts of a Life Event that uses the Citizen Data Hub. This section of the guide 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 IEG script is much like writing any other IEG script for more information on writing IEG2 Scripts in general please refer to the Developer Guide *Authoring Scripts In IEG2* . However there are some special considerations for Life Event scripts. In the main these depend on whether the Life Event is a Round Tripping Life Event or a Standard Life Event. Recall that in a Round Tripping Life Event, Citizen Data is read into

the Data Store used by the IEG script and then this data can be modified by the Citizen as they go from page to page in the Life Event script. Take for example a piece of Income data that is read into the Life Event script. The Citizen modifies this Income information and then submits. The Life Event Broker must ensure that when the Citizen changes the Income data, that this change is detected and that the changes are correctly propagated back to the Income entity from which the data was originally read. 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. In order to facilitate this the IEG script designer must place a "marker" into the data store schema. Here 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. There are some other special markers that can be placed in the schema to aid with providing automatic updates to Cúram evidence entities. These will be discussed in subsequent sections.

When designing a script for a Round Tripping Life Event the designer should bear in mind the effects that pre-population of data can have on the flow of the script. One particular example of this is conditional clusters. Life Event Scripts should avoid conditional clusters that are associated with pre-populated data. These are common in Intake scripts but don't work well when the data store has been pre-populated. Take for example a Life Event around losing 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 is used to drive the display of a conditional cluster that identifies which household members who have jobs. If the data in the data store is pre-populated however, there's a good chance that one or more there will be one or more `Person` entities with `hasJob` already be set to "true". In the current implementation of IEG2 however 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 general rule should be to avoid control questions for conditional clusters like this when the fields they control are pre-populated.

Writing Life Event Review Scripts

Users who have 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. A script that is suitable for data collection however is not necessarily 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 have been already entered. Summary pages are recommended as a good way of writing Life Event Review Scripts. For more information on writing IEG2 Scripts please refer to the Developer Guide *Authoring Scripts In IEG2* . If a review script is not supplied, then the question script is launched 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, readers are referred to the Chapter Customizing Triage in this guide and . 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 correspond 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

This section describes the artifacts that need to be developed in order to pre-populate a Life Event script. In this section we describe

- 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 Temporal 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. The next section demonstrates how to write XSLT Transforms for use in the Data Hub.

Authoring Read Transforms

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>
```

Example 9.1 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="-4160200155783495 68" vehicleMake="VM2"
          versionNo="2" amountOwed="3,200.00" vehicleModel="159"
          year="2008" fairMarketValue="17,000.00"
          curamEffectiveDate="20110301">
            <Descriptor/>
        </Resource>
    </Person>
</Application>
```

Example 9.2 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>
```

Example 9.3 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

This section describes the artifacts that need to be developed in order to process the data submitted from a Life Event script. This section 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>
```

Example 9.4 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

Configuring the Evidence Broker for use with the Holding Case

The Holding Case is of little value by itself, it is simply, as the name implies, a Holding Area for Evidence before it is sent somewhere else. Nor-

mally the goal once data has been updated on the Holding Case, is to broker these updates to Integrated Cases so that Case Workers can vet the changes and apply them to the relevant cases. Once the data is accepted onto the Integrated Cases then James will start to see the positive impact of submitting a Life Event as the updated data can start to have an impact on his benefits. The bridge between the Holding Case and the Integrated Cases can only be crossed if the appropriate Evidence Broker configuration is defined. This section demonstrates how that can be achieved. For background on the Evidence Broker the reader is referred to the Developer Guide: *Cúram Evidence Broker Developers Guide* .

Configuring Sharing from The Holding Case

Below is an example evidence configuration for sharing of Pregnancy evidence from the Holding Case to an Integrated 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>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 sharing from the Holding Case to another Integrated Case, the source type should be *CT10301* and the source id should be set to 10330. The source evidence type should be set to `ET10000` , which is the code for all Evidence 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 identifies Pregnancy Evidence. The evidence sharing type should be set to `SET2002` which is the code for Non-Identical Sharing. Note, that the `AUTOACCEPTIND` is set to 0. It is strongly recommended that this always be set to 0 when sharing from a Holding Case to an Integrated Case. This setting means that a Case Worker will always get to vet any changes that have come in from the Citizen's Holding Case. Assuming the Case Worker 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 containing configuration that follows the example given above. 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 containing the Pregnancy data into a new Pregnancy Evidence Record in the target Integrated Case. Previously this guide has alluded to the fact that Holding Evidence is not "standard" Evidence. In fact it is stored in an XML representation, so in the process of copying the Holding Evidence to the Target Evidence type the Evidence Broker must perform a conversion of the XML data into standard Evidence data. To assist with this conversion process it is necessary to supply meta-data. An example of this meta-data is illustrated below:

```
<?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 meta data describes each of the entities that can be copied from the Holding Case to an Integrated Case and *vice versa* . The meta data 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 doesn't just 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 doesn't exist, then creates one.

This meta data is stored in a an `AppResource` (For more information about `AppResources` , refer to the Cúram Developer Guide `Authoring Scripts in IEG2`). The resource store key is identified by the Cúram Environment Property `curam.workspaceservices.datahub.metadata` . Out of the box the value for this variable defaults to the value `curam.workspaceservices.datahub.metadata` . This points to some default Holding Evidence Data Hub Meta Data. The following steps can be used to replace the default Holding Evidence Data Hub Meta Data 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 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 may also want to consider whether evidence should 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 Case Worker to selected evidence can be propagated back to the Holding Case. This is essential for Life Events that need to pre-populate data from Evidence Entities in existing Integrated Cases. The example below 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 that, unlike Sharing from Holding Case to Integrated Case, the *AUTOACCEPTIND* is set to 1. This is because the target case is a Holding Case and Holding Cases are designed to operate unattended. It is not expected that Case Workers should need to review items being shared onto the Holding Case as they come from an authoritative source, i.e. the Integrated Case.

Issues for Consideration

With suitable configuration, It is possible to share data from the Holding Case to many different Integrated Cases. Imagine that two different Integrated Cases A and B are configured to share information with a James' Holding Case H. Both cases A and B have separately recorded an Income Evidence record for James. In James' Holding Case this will show up as two separate Income Records and as far as cases A and B are concerned they *are* two entirely separate records, A's view of James' Income and B's view of James' Income. To James however this might not make much sense - he has only one Income and is using one Portal to communicate with the SEM or SEMs concerned. Why should he see two records for the same Income? In cases like this, where there is sharing to multiple Integrated Cases from a single Holding Case, consideration should be given to creating another set of sharing relationships should be established from A to B and B to A. This is an issue that will require proper consideration early on in the project Life

Cycle.

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.

Putting it all Together

Previous sections in this chapter have discussed how to create all the constituent pieces of a Life Event, this section 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 *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\`.

9.6 Life Events API Guide

This Section describes how to use the Java API for Life Events and the Citizen Data Hub. This section assumes that the reader is familiar with the material already introduced in this chapter.

9.6.1 Event APIs for Life Events

To understand this section the reader must be familiar with the contents that precede this section in this Chapter. 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 prior to 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);
```

Chapter 10

Universal Access Web Services

10.1 Introduction

This section describes the IBM Cúram Universal Access web services and how to develop peer code to communicate with those web services.

In some scenarios, customers will deploy IBM Cúram 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, IBM Cúram Universal Access can be configured to communicate with various remote systems using web services.

Universal Access supports the following outbound web services:

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

Cúram 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.

10.2 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 Administrator and Universal Access Entry Edition Administrator applications.

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 in Appendix A. 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.

10.3 Process Application Service

10.3.1 Receive Application

This outbound web service is invoked by Universal Access on remote systems. It is used to communicate an application for benefits for one or more social programs. 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 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 should be stored by the receiver.
- applicationLocale – Denotes the preferred locale of the user who entered the application. For example ”en_US”. This information should be stored by the receiver. Remote systems can send a variety of 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 is in XML Schema dateTime format. For example, 2012-05-29T15:34:49.000+01:00.

- programsAppliedFor – This 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 corresponds to the value of the Reference field configured in the Programs section of Universal Access 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 datastore associated with an application.
- applicationSchemaName – The name of the schema 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 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 has been successfully processed. In the case that there is a problem processing the request, a fault must be returned containing a string to indicate the nature of the problem. The String should be localized to the locale of the Universal Access Server since it will appear 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, however, that the receiver should never receive two applications from A with an application reference of 256.

10.3.2 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 localized 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 recognised 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. A sample SOAP request for this web service is published in Appendix A.

10.4 Update Application Service

10.4.1 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. A sample SOAP request for this web service is published in Appendix A.

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 appendix A for sample SOAP message for details). The username placed within the credential must match the Source User Name entered into the Remote System entry corresponding to the peer system sending the request.

10.4.2 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. A sample SOAP request for this web service is published in Appendix A.

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.

Please refer to Appendix A which contains a 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.

10.5 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.

10.6 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 recognise 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. A sample SOAP request for this web service is published in Appendix A.

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 further informaiton on configuring remote systems see the Configuring Remote Systems chapter the 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 via a fault response. Possible issues include:

- An account has already been associated with the intake application reference.
- The username already exists.
- The username and/or password do not meet minimum mandatory criteria for password strength, username length etc.

10.7 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. A sample SOAP request for this web service is published in Appendix A.

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.

10.8 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. A sample SOAP request for this web service is published in Appendix A.

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.

10.9 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 into 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. A sample SOAP request for this web service is published in Appendix A.

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.

10.10 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. A sample SOAP request for this web service is published in Appendix A.

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.

10.11 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. A sample SOAP request for this web service is published in Appendix A.

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.

10.12 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. A sample SOAP request for this web service is published in Appendix A.

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.

Chapter 11

Fully Customizable Universal Access Artifacts

11.1 Introduction

This chapter describes the artifacts that are fully customizable in Universal Access and how to go about making any customization to these artifacts.

11.2 Customizable Universal Access Page Content

Universal Access public pages are driven by page player XML pages that are made available in the app resource store. Each page has a corresponding property file and images used in rendering these page 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.

11.2.1 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.

11.2.2 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.

11.2.3 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 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.

Please see the chapter on customizing citizen account for information regarding adding new languages to citizen account pages.

11.2.4 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.

Please see the chapter on customizing citizen account for information regarding customizing the look and feel of citizen account pages.

11.2.5 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.

11.3 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 via Event and Strategy patterns as described in the Cúram Development Compliancy Guide.

11.4 Extendable Code Tables

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

Chapter 12

Universal Access Artifacts with Limited Scope for Customization

12.1 Introduction

This chapter describes the restricted access artifacts present in Universal Access. Customers that are looking to change artifacts discussed in this chapter should consider alternatives or request an enhancement to Universal Access.

12.2 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/or upgrade issues. This applies to the model files in the following packages:

- WorkspaceServices
- CitizenWorkspace
- CitizenWorkspaceAdmin

12.3 Universal Access Page Player XML

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

12.4 JSP and JSPX pages

Customers are not supported in making changes to any out of the box jsp or

jspx files.

12.5 Javascript files

Customers are not supported in making changes to any out of the box javascript files in the following components:

- WorkspaceServices
- CitizenWorkspace
- CitizenWorkspaceAdmin

12.6 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

12.7 Client-side Java artifacts

Universal Access delivers all of its client-side Java artifacts via 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.

12.8 Code Tables

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

Appendix A

Sample SOAP Requests

A.1 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>
```

A.2 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>
```

A.3 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>
```

A.4 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>
```

A.5 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>
```

A.6 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>
```

A.7 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>
```

A.8 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>
```

A.9 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>
```

A.10 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>
```

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing

Legal and Intellectual Property Law.

IBM Japan Ltd.

1623-14, Shimotsuruma, Yamato-shi

Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typograph-

ical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Dept F6, Bldg 1

294 Route 100

Somers NY 10589-3216

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources.

IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products

should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

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

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/us/en/copytrade.shtml .

Adobe, the Adobe logo and Portable Document Format (PDF), are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Apache is a trademark of Apache Software Foundation.

WebLogic Server, Java and all Java-based trademarks and logos are registered trademarks of Oracle and/or its affiliates.

Other names may be trademarks of their respective owners. Other company, product, and service names may be trademarks or service marks of others.