IBM Cúram Social Program Management

Cúram Server Developer's 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.

Table of Contents

Chapter 1

Introduction

1.1 Introduction

The Server Development Environment (SDEJ) of *IBM® Cúram Social Program Management* enables the development of high-quality, low-cost client server applications through model driven generation. This generation facilitates client-server development by taking a Unified Modeling Language (UML) model and producing the following:

- Generated *Java®* code;

- Data Definition Language (DDL) describing the database entities in the model, enabling instances of a database to be defined in a human and machine readable form;

- Support for remote invocation

The *Cúram Solution Architecture* document provides an introduction to the Cúram Generator and its outputs. While the fundamental elements of a server application are supplied by the Cúram Generator, certain custom coding and configuration tasks must be performed.

The *Cúram Security Handbook* document should be referenced for all aspects of security that must be considered when developing and deploying a Cúram enterprise application, e.g. authentication and authorization of users.

This document describes how to develop the custom code in Cúram server applications, and how to build the resultant applications. It is a reference guide that should be read by programmers wishing to develop custom code for Cúram server applications using the SDEJ. It is not intended as an introductory document, or as guide on how to deploy a Cúram application on an Application Server[1] (this is described in the *Cúram Deployment Guide* for the appropriate application server).

1.2 Content Summary

This guide provides details on a number of topics which can be grouped under three main headings:

- Building and Configuring a Cúram Application

- SDEJ Development and Application Programming Interfaces

- Cúram Runtime Behavior

- Building and Configuring a Cúram Application

  - Chapter 2, *Directory Structure* provides an introduction to the layout of the application.

  - Chapter 3, *Build Files and Their Targets* details the build support provided.

  - Chapter 4, *Cúram Configuration Settings* enumerates the various configuration settings supported by the infrastructure.

  - Chapter 5, *Data Manager* details the Data Manager—a tool that can be used to create a database to support the Cúram application.

  - Chapter 6, *SQL Checker* details the SQL Checker—a tool that can be used to ensure the semantic and syntactic correctness of SQL which has been hand-crafted by an Application Developer.

- SDEJ Development and Application Programming Interfaces

  - Chapter 7, *Eclipse* describes relevant aspects of *Eclipse* usage, as well as providing some tips and tricks.

  - Chapter 8, *Logging* details the infrastructure support for the logging/tracing of status and error information.

  - Chapter 9, *Using Exceptions* details the infrastructure support for the creation, tracing and display of exceptions.

  - Chapter 10, *Message and Code Table Files* details the format of the message files and code table files that are used within Cúram.

  - Chapter 11, *Specialized Readmulti Operations* explains the usage of Specialized Readmulti Operations which can be used to replace standard readmulti operations with specialized processing.

  - Chapter 12, *Deprecation* describes deprecation in Cúram: what it is, how it can affect custom code, what it means for support and the associated build infrastructure that helps pinpoint custom artefact dependencies on deprecated Cúram artefacts.

  - Chapter 13, *User Preferences* describe how to define and use User

Preferences for a Cúram application.

- Cúram Runtime Behavior

  - Chapter 14, *Transaction Control* details the aspects of Transaction Control within a Cúram application that must be understood by a developer.

  - Chapter 15, *Transaction SQL Query Cache* outlines the details of a cache that can store the results of any SQL queries that do a SELECT on a database table for the duration of the transaction in which the operation was invoked.

  - Chapter 16, *Deferred Processing* describes how to achieve deferred processing in a Cúram application

  - Chapter 17, *Timer Bean* describes the functionality that allows timers to be defined to invoke client-visible methods at a specified time.

  - Chapter 18, *Events and Event Handlers* describes Events, a mechanism for loosely-coupled parts of the Cúram application to communicate information about state changes in the system.

  - Chapter 19, *Unique IDs* details the infrastructure support for Unique Identifiers—numbers generated by the Cúram infrastructure for use as unique database keys.

1.3 Overview of Compliant Development Artifact Changes

Aside from your new custom development (e.g. adding message files, code tables, events, etc.) you may also need to modify Cúram out-of-the-box (OOTB). The following summarizes the range of compliant changes you can make to the out-of-the-box Cúram development artifacts:

Type of Change	Initial Artifact(s)	Reference
Change an existing message file	Message file (externalized server informational, warning, and error messages - `.xml` files in the `message` directory)	Section 10.2, *Message Files*
Remove an existing message	Message file (externalized server informational, warning, and error messages - `.xml` files in the `message` directory)	Section 10.2, *Message Files*
Add additional loc-	Message file (externalized	Section 10.2.6, *Localizing*

Type of Change	Initial Artifact(s)	Reference
ale (i.e. language) support to an existing message	server informational, warning, and error messages - `.xml` files in the `message` directory)	*SDEJ Message Files*
Change an existing code table name	Code Table file (code value pairs - `.ctx` files in the `codetable` directory)	Section 10.3, *Code Table Files*
Add a new code table item into an existing code table	Code Table file (code value pairs - `.ctx` files in the `codetable` directory)	Section 10.3, *Code Table Files*
Change the description of an existing code table	Code Table file (code value pairs - `.ctx` files in the `codetable` directory)	Section 10.3, *Code Table Files*
Disable an existing code table item	Code Table file (code value pairs - `.ctx` files in the `codetable` directory)	Section 10.3, *Code Table Files*
Remove an existing code table item	Code Table file (code value pairs - `.ctx` files in the `codetable` directory)	Section 10.3, *Code Table Files*
Add additional locale (i.e. language) support to an existing code table	Code Table file (code value pairs - `.ctx` files in the `codetable` directory)	Section 10.3.6, *Localizing SDEJ Code Table Files*
Add an event registration (to augment Cúram OOTB functionality)	Event Definition file (`.evx` files in the `events` directory) & Event Handler Registration file (`handler_config.xml` in the `events` directory)	Chapter 18, *Events and Event Handlers*
Disable an existing event handler	Event Definition file (`.evx` files in the `events` directory) & Event Handler Registration file (`handler_config.xml` in the `events` directory)	Chapter 18, *Events and Event Handlers*
Override an existing user preference	User Preference file (`DefaultPrefer-`	Section 13.2, *User Preferences Definition*

Type of Change	Initial Artifact(s)	Reference
	`ences.xml` file in the `userpreferences` directory	
Override an existing application property	Application Property File (`Application.prx` file in the `properties` directory)	Section 4.2.1, *Application.prx*
Add initial, demo or test data data (rows) to an existing database table	DMX File (script for populating the database with data - `.dmx` files in the relevant `data` subdirectory)	Section 5.4.2, *Data Contents DMX Files*

Table 1.1 Cúram Development Artifact Compliant Changes

Notes

[1]*IBM® WebSphere® Application Server* and *Oracle® WebLogic* are the supported application servers. For exact information on versions, please refer to the *Cúram Supported Prerequisites* document.

Chapter 2

Directory Structure

2.1 Overview

The directory structure for the server side *IBM Cúram Social Program Management* application, and the underlying Server Development Environment (SDEJ) are described in this chapter.

2.2 Application Components

2.2.1 Component Folders

The Cúram server application is organized into collections of artifacts called *components*. Each component has its own folder below the `<EJBServer>/components` folder. The `core` component is always present. This contains all of the artifacts needed for the core functionality of the Social Program Management Platform. The name of the component folder is used as the name of the component.

2.2.2 Component Order

There can be any number of application components, but they are processed in a strict *component order*. This order determines the priority that will be given to artifacts that share the same name but appear in different components. This is fundamental to the manner in which server artefacts are customized.

The component order is defined by the `SERVER_COMPONENT_ORDER` environment variable. This is a comma-separated list of component names. Use only commas; do not use spaces. You must place the component with the highest-priority first in the list and continue in descending order of priority. The `core` component always has the lowest priority and is implicitly assumed to be at the end of the list; you do not need to add it explicitly.

For example, setting the component order to "MyComponentOne,MyComponentTwo" will give the highest priority to artifacts in the `MyComponentOne` folder within `<EJBServer>/components`, a lower priority to artifacts in the `My-ComponentTwo` folder, and the lowest priority to artifacts in the `core` folder. Any component folder not listed in the component order will not will automatically be added to the end of the component order in alphabetical order. If you do not set the component order at all, the default component order will include all components in alphabetical order.

Localized Components

Localized components contains translated artifacts for the base components and are of the format "<component name>_<locale>". It is *not* necessary for these to be added to the `SERVER_COMPONENT_ORDER` environment variable as the tooling that processes this environment variable will prepend any available components that match entries in the `SERVER_LOCALE_LIST` environment variable. Localized components are matched both on complete locale entry and on the two-character, lower-case language code. Localized components are prepended before the base component in the complete component order.

2.3 Application Directory Structure

Two aspects of the Cúram application directory structure are described; the structure related to the source artefacts associated with an application, and the resultant structure when the application is built.

2.3.1 Source Artefacts of the Cúram Application

Example 2.1, *Cúram Application Structure* shows the directory structure for the source artefacts of a Cúram application project i.e. the structure prior to performing a build. Table 2.1, *Cúram Application Installation Structure* describes each directory within the directory structure in more detail.

Example 2.1, *Cúram Application Structure* includes the `SERVER_DIR`, `ProjectPackage` and `CodePackage` as place holders.

- The `SERVER_DIR` is the root of the server directory structure; the location of the `EJBServer` directory within the Cúram application.

- The `ProjectPackage` is a global setting, set at build time. It is set to `Cúram` in the reference application which is shipped with Cúram.

- The `CodePackage` is based on a model setting which is described in the *Cúram Modeling Reference Guide*. It allows individual components to be scoped within their own logical packages. Any number of Code Packages may be nested inside each other.

```
SERVER_DIR
  + project
    + config
    + properties
  + components
    + core
      + codetable
      + data
      + doc
      + events
      + lib
      + message
      + model
      + properties
      + rulesets
      + sample
      + webservices
      + workflow
      + wsdl
    + custom
      + source
        + <Project Package>
          + impl
        + <Code Package>
          + impl
            + wsdl
  + build.bat
  + build.sh
  + build.xml
  + buildhelp.bat
  + deprecationreport.xml
  + .classpath
  + .project
```

Example 2.1 Cúram Application Structure

Name	Contents
project	A top level directory containing all information that is relevant to the entire project rather than specific components.
project/config	Configuration information related to the project, including top level configuration files for the data manager and web services connector.
project/properties	Properties that relate to the project as a whole.
components	Each project is made up of a number of components. This directory is simply a place holder for those components.
components/core	A pre-defined component which is used by all other components.
components/core/codetable	Codetable XML (ctx) files created by the developer are kept here. These files are used to define codetables for a Cúram application. The outputs produced from a codetable file consist of an SQL script to populate the code table in the database, and a *Java®* file which provides the necessary constants to the application. See

Name	Contents
	Chapter 10, *Message and Code Table Files* for more information
components/core/data	The Data Manager for this component.
components/core/doc	The JavaDoc for this component.
components/core/events	Event XML (`evx`) files created by the developer are kept here. These files are used to define event classes and event types for a Cúram application. The outputs produced from an event file consist of an SQL script to populate the event class and event type tables in the database, and a *Java* file which provides the necessary constants to the application. See Chapter 18, *Events and Event Handlers* for more information
components/core/lib	Contains the built component code packaged in a jar e.g. core.jar.
	Additionally, any third-party jar files specified here will automatically be included in the classpath used during compilation or a Batch Launcher run. Files listed here will also be added to any EAR (Enterprise ARchive) file created and an entry added to the manifest file to reference this file.
components/core/message	Message (`.xml`) files created by the developer are stored here. The *Java* artefacts produced from a message file are a *Java* file and a properties file. See Chapter 10, *Message and Code Table Files* for more information
components/core/model	The elements of a Cúram application UML model that relate to this component are available here.
components/core/properties	The component specific Application property definitions are stored here.
components/core/rulesets	Rules (`.xml`) files created by the developer are stored here. These files may be hand-crafted or created via an online client (Rules Editor). The *Cúram Rules Editor Guide* describes how to create these files and the *Cúram Rules Definition Guide* explains ruleset structure in some detail.
components/core/sample	An optional directory containing a zip file of a set of sample java source files matching the component built code within the lib directory. Used for debugging or reference,

Name	Contents
components/ core/webservices	An optional directory containing the `.xsd` schema files that are referenced by web services in this component.
components/core/workflow	Workflow process definition (`.xml`) files created by the developer are stored here. These files may be hand-crafted or created via an on-line client (Process Definition Tool). The *Cúram Workflow Reference Guide* describes these files in some detail.
components/core/wsdl	An optional directory containing the `.wsdl` (Web Service Description Language) files that are invoked from this component. A WSDL description can be spread over several files that reference each other possibly in some arbitrary directory structure. These references can be resolved as long as they are relative
components/custom	Any number of new components may be added. They all have the same structure as the core component.
components/custom/source	All handcrafted *Java* source code, produced by the developer, is located here.
build.bat, build.sh	A command file that builds your project. This wraps the `build.xml` file (an *Apache Ant* build file) that is contained within the `EJB-Server`. The build structure and use of this file is described in Chapter 3, *Build Files and Their Targets*
build.xml	An *Ant* build file that extends the SDEJ build scripts to enhance a number of targets.
deprecationreport.xml	An *Ant* build file that provides deprecation reporting.
buildhelp.bat	A command file that displays project help. This wraps the `build.xml` file. The use of this file is described in Chapter 3, *Build Files and Their Targets*.

Table 2.1 Cúram Application Installation Structure

2.3.2 Cúram Application Build Structure

This section describes the directory structure created when a Cúram application is built. Example 2.2, *The Cúram Application Build Structure* presents the new directories that are created during the build process while Table 2.2, *Build Directory Structure* gives more details on the contents of each direct-

ory.

```
SERVER_DIR
  + build
    + datamanager
    + ear
      + WAS
      + WLS
    + jar
    + sqlcheck
    + svr
      + cls
      + codetable
        + cls
        + gen
        + scp
        + sql
      + events
        + cls
        + gen
        + scp
      + gen
      + message
        + cls
        + gen
        + scp
      + webservices
      + workflow
      + wsc
      + wsc2
  + buildlogs
```

Example 2.2 The Cúram Application Build Structure

Name	Contents
build/datamanager	Contains intermediate files produced by the Data Manager and the resulting merge dmx files from the initial, demo and test directories. The Data Manager creates the intermediate files when translating the database independent files into a format which can be loaded onto the database. Five database dependent `.sql` files are produced as well as one database independent `.xml` file which is responsible for loading the Large OBjects (LOBs) onto the database.
build/ear/WAS	The `.ear` file produced for *WebSphere Application Server*.
build/ear/WLS	The `.ear` file produced for *WebLogic*.
build/jar	Jar files created by the command line project build.
build/sqlcheck	A database dependent `sqlj` file which contains a subset of the dynamic SQL statements from the model and the inserts from the Data Manager collated together.
build/svr	All build artefacts for the server side.
build/svr/cls	All of the compiled class files for the applica-

Name	Contents
	tion.
build/svr/gen	Generated server side sources.
build/svr/gen/ddl	Database independent definition scripts that establish the structure of a Cúram server application's database tables are generated into this directory. Some intermediate files (including a representation that is used to build to database dependent `sqlj` file) are also generated into this directory.
build/svr/gen/<ProjectPackage>	Root of the generated server source code hierarchy.
build/svr/gen/int	Intermediate files produced during the build.
build/svr/codetable/cls	The compiled codetable files.
build/svr/codetable/gen	The generated codetable file artefacts.
build/svr/codetable/scp	A copy of the results of merging the individual codetable files according to the component order (`SERVER_COMPONENT_ORDER`).
build/svr/events/cls	The compiled event class and event type files. These may be used as constants in the Cúram application.
build/svr/events/gen	The generated events file artefacts which include the `.java` files containing the event class and event type constants and `.dmx` files to be used to populate the event class and event type tables on the database.
build/svr/events/scp	A copy of the results of merging the individual event files according to the component order (`SERVER_COMPONENT_ORDER`).
build/svr/message/cls	The compiled message files.
build/svr/message/gen	The generated message file artefacts.
build/svr/message/scp	A copy of the results of merging the individual message files according to the component order (`SERVER_COMPONENT_ORDER`).
build/svr/webservices	Compiled class files for the web service support elements of the application.
build/svr/workflow	A copy of the results of determining the individual workflow process definition files to be loaded onto the database according to the component order (`SERVER_COMPONENT_ORDER`).
build/svr/wsc2	Compiled class files for the *Apache Axis2-*

Name	Contents
	generated client stubs for each registered outbound web service connector.
<app.xml>	Extracted UML model contents, named as per model.
buildlogs	A log file is created each time a build is performed and is stored here. This can be used to investigate any problems with the build process.

Table 2.2 Build Directory Structure

2.4 Artefacts of the SDEJ

Example 2.3, *SDEJ Structure* shows the directory structure of the SDEJ when installation is complete, while Table 2.3, *SDEJ Structure at Installation* gives more details on the contents of each directory. The CURAMSDEJ is the root of the directory structure; the name given to wherever the SDEJ has been set up or installed.

```
CURAMSDEJ
+ bin
+ codetable
+ doc
+ drivers
+ ear
+ lib
+ message
+ rsa
+ scripts
+ util
+ xmlserver
```

Example 2.3 SDEJ Structure

Name	Contents
bin	This directory contains all *Ant* build scripts necessary to build, verify and configure a Cúram application. The build.bat script file delivered with the Cúram application hooks into this directory to invoke the build.xml file contained here. Use of this file is described in Chapter 3, *Build Files and Their Targets*
codetable	This directory contains the set of codetable files shipped by the SDEJ. These files use the file extension .itx. Each of these files can be customized, see Section 10.3.6, *Localizing SDEJ Code Table Files* for more details.
doc	This directory contains the JavaDoc shipped

Name	Contents
	with the SDEJ.
drivers	This directory contains the drivers used by the SDEJ to access the database.
ear	This directory contains the deployment descriptors and templates necessary to build application ear (Enterprise Archive) files for the chosen application server.
lib	This directory contains the compiled SDEJ source, Third Party JAR files, XML schemas and stylesheets necessary to fulfill all SDEJ functionality.
message	This directory contains the set of message files shipped with the SDEJ. Unlike the Cúram application message files these infrastructure message files use the file extension `.iml`. Each of the files can be customized, See Section 10.2.6, *Localizing SDEJ Message Files* for more details.
rsa	This directory contains the *Eclipse* plugin artefacts used to provide Cúram functionality in *IBM® Rational® Software Architect*. See the *Working with the Cúram Model in Rational Software Architect* for more details.
scripts	This directory contains the database independent XML files necessary to create the database required by the SDEJ.
util	This directory contains useful utilities shipped with the SDEJ.
xmlserver	This directory contains the artefacts and build scripts necessary to run the xmlserver. See *Cúram XML Infrastructure Guide* for more information

Table 2.3 SDEJ Structure at Installation

Chapter 3

Build Files and Their Targets

3.1 Overview

The *IBM Cúram Social Program Management* Server Development Environment (SDEJ) uses *Ant* to process its build files. These *Ant* build files are located in the `/bin` directory of the SDEJ. The build files are invoked through `build.bat` and `buildhelp.bat`.

This chapter explains how to build a Cúram application once it has been installed, and the optional parameters which can be provided when performing a build.

3.2 Performing the Build

Invoking `buildhelp` at the command line (in `SERVER_DIR`) will show all available targets. A single build target is required to build the Cúram application out-of-the-box for development. The user should:

- Start a command prompt and change directory to the top level of the Cúram project; the `SERVER_DIR`.

- Set up any environment variables that were not set as system properties during the installation process as described in the *Cúram Third Party Tools Installation Guide* (e.g., `JAVA_HOME`, `J2EE_JAR` and `ANT_HOME`).

- Set up `SERVER_DIR` to point to the top level of your Cúram project.

- Set up `SERVER_MODEL_NAME` to be the name of your Cúram project.

- Type **build server** and hit return to invoke this build target.

Invoking `buildhelp` at the command line (in `SERVER_DIR`) will show all available targets. A single build target is required to build the Cúram application out of the box for development. The user should:

16

3.3 Overriding default JUNIT JAR

The `junit.jar` file is set by default relative to the `JUNIT_HOME` environment variable, e.g. `${sysenv.JUNIT_HOME}/junit.jar`. To override the location/naming of the `junit.jar` file, a new system property `JUNIT_JAR` is available for this purpose. If the `JUNIT_JAR` system property is set, this will take precedence over the default. An example of its usage (e.g. *Microsoft® Windows*): set `JUNIT_JAR=c:\junit-4.8.jar`

3.4 Configuring the Build

This section describes the optional parameters that can be provided when building the Cúram application.

3.4.1 Cúram Build Settings

A number of parameters may be passed when performing the build. They should be passed in the following way **build server -Dsome.setting=somevalue**. These parameters are:

Parameter	Values	Description
dir.sde	directory name	The name of the directory containing the installed SDEJ that you want to use for this build. The default is the directory referred to by the `CURAMSDEJ` environment variable.
prp.loglevel	info warn error verbose debug	The logging level used when recording build progress to the build log. The default is *info*.
prop.file.location	directory name	Override the location of the directory that is used to pick up the property files. By default the `<ProjectName>/properties` directory is used.
prp.maxcodetable codelength	number	Override the maximum length of a code table `code`. This is used for validation of codetables during generation, where it is desired to ensure that the code length defined in the codetables being generated do not exceed the length specified. This is

Parameter	Values	Description
		to ensure, you catch errors before entering codetables onto the database. This does not override the maximum length on the database *.
prp.maxcodetable namelength	number	Override the maximum length of a code table `name`. This is used for validation of codetables during generation, where it is desired to ensure that the name length defined in the codetables being generated do not exceed the length specified. This is to ensure, you catch errors before entering codetables onto the database. This does not override the maximum length on the database *.
prp.maxcodetable descriptionlength	number	Override the maximum length of a code table `description`. This is used for validation of codetables during generation, where it is desired to ensure that the description length defined in the codetables being generated do not exceed the length specified. This is to ensure, you catch errors before entering codetables onto the database. This does not override the maximum length on the database *.
prp.warningstoerrors	true false	Indicates that warnings thrown when extracting and generating from the model, code table and message files should be treated as errors (an error typically terminates the process). The default is `false`.
prp.forcegen	"-force:modelgen"	Indicates that the build should progress even if errors are found when generating code from the model. The default is that this should not occur. This means that if this flag is set and errors are found during generation, the build is not interrupted after the **modelgen** build target is executed. Once this target is complete it will eventually pass onto the **compile.generated** target. See Sec-

Parameter	Values	Description
		tion 3.5, *What is happening under the hood?* for more details.
		Note: The errors are still reported.
prp.noninternedstrings	true false	Indicates whether code table artefacts should be generated with strings which will not be interned. This is described in more detail in Section 3.5.1.5, *ctgen*. The default is `true`.
curam.using.dbcs	true false	Should be set if the Cúram model contains DBCS (Double Byte Character Set) characters. If defined the Cúram application model is first processed by the utility native2ascii. The Model Extractor then uses this new reworked model to produce <project>.xml file. If this property is not specified the Model Extractor takes original model file as its input.
curam.using.nonascii	true false	Should be set if the Cúram model contains non ascii characters. If defined the application model is first processed by the utility native2ascii. The Model Extractor then uses this new reworked model to produce <project>.xml file. If this property is not specified the Model Extractor takes original model file as its input.
extra.generator.options	String	Specifies additional command line parameters for the server code generator. These settings are described in Section 3.4.4, *Generator Settings*.
portability.warnings	BUILD, DMX	Specifies whether the SQL Checker should determine if the build is portable to other database platforms and whether the Data Manager files are valid. The default is to check all of these.
enablefacade	true false	Specifies that the build should generate the session beans and their corresponding deployment artefacts for model elements identified as facades. The default is `false` which means they will not be generated.

Parameter	Values	Description
prp.genschemaval idation	true false	Indicates that the .xml file produced by the model extractor will be validated against a schema when it is being parsed and used by the code generator to generate the application code. The default is `false`.

Table 3.1 Build Configuration Settings

ℹ️ Database update for code table property changes

The relevant database column lengths must be altered to support the changes made by using the prp.maxcodetablecodelength, prp.maxcodetablenamelength, or prp.maxcodetabledescriptionlength properties.

The columns should be altered using the Data Manager. In each case a handcrafted SQL script that alters the relevant column's length should be added to the custom database scripts folder. This script should then be added as an entry to the `dataman-ager_config.xml` file before loading the code tables into the database. Please refer to Chapter 5, *Data Manager* for further information on using the Data Manager.

3.4.2 Java Compiler Settings

The following parameters may be passed when performing the build and control the behavior of the *Java* compiler. They should be passed in the following way **build server -Dcmp.debug=on**. These settings are:

Parameter	Values	Description
cmp.debug	on off	Indicates whether the source should be compiled with debug information. The default is `on`.
cmp.maxmemory	Number	The maximum size of the memory for the underlying VM. The default is `768`.
cmp.nowarn	on off	Indicates whether the `-nowarn` switch should be passed to the compiler. The default is `off`.
cmp.maxwarnings	Number	Asks the compiler to set the maximum number of warnings to print. The default is `10000`.
cmp.optimize	on off	Indicates whether source should be compiled with optimization The default is `off`.

Parameter	Values	Description
cmp.deprecation	on off	Indicates whether source should be compiled with deprecation information. The default is `off`.
cmp.verbose	true false	Asks the compiler for verbose output. The default is `false`.
cmp.include.AntRuntime	yes no	Indicates whether the *Ant* run-time libraries should be included on the classpath. The default is `yes`.
cmp.include.JavaRuntime	yes no	Indicates whether the default run-time libraries, from the executing VM (Virtual Memory), should be included on the classpath. The default is `no`.
cmp.failonerror	true false	Indicates whether the build will continue even if there are compilation errors. The default is `true`.
cmp.listfiles	yes no	Indicates whether the source files to be compiled will be listed. The default is `no`.
PRE_CLASSPATH	Filename	An environment variable to allow jar files to be added to the start of the classpath used during compilation or a Batch Launcher run. Files listed here will be added to any EAR (Enterprise ARchive) file created and an entry added to the manifest file to reference this file. Files should be separated with the relevant Path separator for your operating system.
POST_CLASSPATH	Filename	An environment variable to allow jar files to be added to the end of the classpath used during compilation or a Batch Launcher run. Files listed here will be added to any EAR file created and an entry added to the manifest file to reference this file. Files should be separated with the relevant Path separator for your operating system.

Table 3.2 Java Compiler Settings

3.4.3 Java Task Settings

The following parameters may be passed when performing the build and control the behavior of the *Java* runtime used by the build scripts. They should be passed in the following way **build server -Djava.fork=true**. These settings are:

Parameter	Values	Description
java.fork	true false	Specifies whether any external classes are executed in another VM. The default is `true`.
java.maxmemory	Number	The maximum size of the memory to allocate to the forked VM. The default is `768m`.
java.failonerror	true false	Specifies whether the build process should be stopped if an external java command exits with a return code other than 0. The default is `true`.
java.jvmargs	String	Specifies the arguments to pass to the forked VM The default is `the empty string`.

Table 3.3 Java Task Settings

3.4.4 Generator Settings

The following parameters may be passed when performing the build and control the behavior of the Cúram Generator. These parameters should be passed in the following way **build server -Dextra.generator.options=-setting1 -setting2**.

These settings are:

Option	Meaning
-nomessage <nnnnn>	Prevent the message with this number from being displayed or acted upon. Note that this can be used to suppress errors which would normally cause the generator to terminate. Doing so can cause the generator to behave unpredictably or produce code which cannot be built.
-primarykeyconstraintprefix <prefix>	Specify a prefix to be applied to primary key constraint names in *IBM® DB2®* and *Oracle® Database*. See the *Cúram Modeling Reference Guide* for more details.
-primarykeyindexprefix <prefix>	Specify a prefix to be applied to primary key index names in *DB2*.

Option	Meaning
	See the *Cúram Modeling Reference Guide* for more details.
-progresslevel <n>	Specify the level of progress to be reported by the generator.
-nonamedprimarykeyconstraint	Specify that names should not be provided for the primary keys. This is off by default i.e. primary keys are named. See the *Cúram Modeling Reference Guide* for more details.
-nonamedforeignkeyconstraint	Specify that names should not be provided for the foreign keys. This is off by default, i.e., foreign keys are named. See the *Cúram Modeling Reference Guide* for more details

Table 3.4 Generator Settings

3.4.5 Other Environment Settings

If you are building on *Red Hat Linux* you may get this error during compilation:

```
unmappable character for encoding UTF8
```

This is due to an encoding mismatch between *Windows* and *Linux* and can be worked around by setting the LANG environment variable as follows:

```
LANG=en_US.ISO-8859-1
```

3.5 What is happening under the hood?

While building the application is as simple as invoking the default target listed above, it is useful for the reader to understand the steps that are involved. Each of these are **ant** targets which may be invoked separately:

3.5.1 generated

This target generates and compiles the code for use in an IDE and wraps the following targets:

- **wsconnector** step generates client stub connectors for outbound web services from .wsdl (WSDL is an acronym for Web Service Definition Language) files registered in the configuration file, <SERVER_DIR>/project/config/webservices_config.xml.

- **wsconnector2** Generates client stub connectors for outbound *Axis2* web

services from the registered WSDL files.

- **emx2xml** - this extracts an intermediate XML representation from a Cúram application UML model.

- **modelgen** - this generates source code and other artefacts from the XML representation of a Cúram application model. It also deletes any artefacts that are no longer represented in the model.

- **msggen** - this merges the message file definitions according to the component order and generates source code and properties from the resultant message definitions.

- **ctgen** - this merges the code table definitions according to the component order and generates source code from the resultant code table definitions.

- **evgen** - this merges the event definitions according to the component order and generates source code from the resultant event definitions.

- **compile.generated** - this compiles any generated source code that doesn't depend on the `impl` directory.

wsconnector

The **wsconnector** step generates client stub connectors for outbound web services from `.wsdl` files registered in the configuration file, `<SERVER_DIR>/project/config/webservices_config.xml`.

An example is shown in Example 3.1, *Example Web Services Configuration*

```
<services>
  <service location=
    "components/<component_name>/wsdl/some_service/TopLevel.wsdl"
  />
</services>
```

Example 3.1 Example Web Services Configuration

The location attribute is the location of the top level WSDL file relative to the `SERVER_DIR`. This configuration file also gives the ability to turn a particular Web Service Connector on and off at will (bearing in mind that business code that accesses the connector would be affected by this). It is acceptable to have no service elements in this file.

The generated connector client stubs must *not* be treated as source. They are intended to be overwritten during each build, based on the WSDL files provided, to ensure the connectors are always synchronized with the web services they represent.

emx2xml

The **emx2xml** step transforms the UML model (which is located in the `<SERVER_DIR>components/*/model` directory) into an intermediate

XML representation. The intermediate representation is stored at the top level of the directory tree.

modelgen

The `modelgen` step transforms the intermediate XML representation into the final *Java* code, deployment support artefacts, web service support artefacts and a set of Data Definition XML files.

Data Definition XML Files

The Data Definition XML files are placed in the `build/svr/gen/ddl` directory and are typically made up of a number of files:

- `<SERVER_MODEL_NAME>_Tables.xml`

- `<SERVER_MODEL_NAME>_Indices.xml`

- `<SERVER_MODEL_NAME>_PrimaryKeys.xml`

- `<SERVER_MODEL_NAME>_UniqueConstraints.xml`

- `<SERVER_MODEL_NAME>_ForeignKeys.xml`

- `<SERVER_MODEL_NAME>_Batch.xml`

- `<SERVER_MODEL_NAME>_Fids.xml`

- `<SERVER_MODEL_NAME>_FieldsReturned.xml`

- `<SERVER_MODEL_NAME>_SQLJ.xml`

The first five of these files contain database independent definitions for creating tables on the database and placing constraints on these tables. `<SERVER_MODEL_NAME>_Batch.xml` describes the persistent data that is necessary to support the batch process related information that has been captured in the UML model. `<SERVER_MODEL_NAME>_fid.xml` describes the persistent data that is necessary to support the security related information that has been captured in the UML model. `<SERVER_MODEL_NAME>_FieldsReturned.xml` describes the persistent data that is necessary to support Field Level Security. `<SERVER_MODEL_NAME>_SQLJ.xml` contains a representation of all the hand-crafted SQL in the model and is used by the checksql target. More information on the contents of these files is provided in Chapter 5, *Data Manager*.

> **ⓘ Foreign Keys and Cúram**
>
> The Cúram application is responsible for enforcing referential integrity and foreign keys are generated to support testing of this. The use of declarative referential integrity (foreign keys) in a production system will impact the performance of that system and is consequently not supported.

Java Code

A large number of *Java* code artefacts are generated as part of this model generation build. They are generated according into a number of categories (and are all located under the `/build/svr/gen/<ProjectPackage>/` and `/build/svr/gen/<ProjectPackage>/<CodePackage>` director- ies). A `CodePackage` may be empty or there may be a number of `Code-Package` elements within each other (for example, `<ProjectPackage>/intf` and `<ProjectPackage>/<CodePackageA>/<CodePackageB>/intf` may both be generated depending on the options that have been chosen).

- `intf` - Defines the interface for the objects that have been modeled.

- `fact` - Provides factory wrappers for the objects identified in bizinter- face.

- `base` - Ensures the developer provides implementations for those meth- ods which must be hand crafted.

- `remote` - Provides remote interfaces for the objects which can be ex- posed to the client.

- `struct` - Defines the classes which model parameters between the ob- jects.

- `rules/rdo` – Defines the classes for the rules data objects. RDOs cannot be stored in code packages so the rules folder is always at the top level. As well as the classes this directory contains a file named `rdoindex.properties` which contains a listing of all the generated objects.

Deployment Artefacts

A number of deployment artefacts are also generated by the model build. This section does not attempt to detail the meaning of these files but simply introduces the files and their locations. These artefacts are used when build- ing an application `.ear` file where they are passed into the `XDoclet` tool. They are generated according to the following categories:

- **IBM Specific Metadata:** provides support for deployment on *Web- Sphere Application Server*. These artefacts are generated into the `/build/ear/WAS`directory and contain the necessary `.xml`, `.xmi` and policy files.

- **Oracle Specific Metadata:** provides support for deployment on *WebLo- gic*. These artefacts are generated into the `/build/ear/WLS` direct- ory and contain the necessary `.xml` files .

Web Service Artefacts

Finally a number of Web Service artefacts are generated. This section does not attempt to detail the meaning of these files but simply introduces the files and their locations. These artefacts are used when building an `.ear` file that supports Web Service invocation. The artefacts consist of special structs which contain web service conversion routines and a web service configuration file (`server-config.wsdd`) and are generated into the `/build/svr/gen/webservices` directory.

msggen

Cúram message files allow a Cúram application to be localized without needing manipulation of hand-crafted code. These files should be used in preference to hard-coded strings within hand-crafted code.

Message files are located in the `/message` directory of a component. The Social Program Management Platform is shipped with a set of message files. These files may be overridden by placing new message files in the `SERVER_DIR/components/<custom>` directory, where `<custom>` is any new directory created under components that conforms to the same directory structure as `components/core`. The override process involves merging all message files of the same name according to a precedence order where the order is based on the `SERVER_COMPONENT_ORDER` environment variable. This variable lists the components in a delimited list in order of priority from most to least important.

The **msggen** build target performs the merge of message files and then translates the resultant merged message file (which is stored in `/build/svr/message/scp` directory) into *Java* source code and property files so it can be accessed at runtime.

The generated *Java* code is then compiled and packed into `/build/jar/messages.jar`.

ctgen

Cúram code table files allow an application to use a level of indirection when storing commonly used constants on the database. This level of indirection enables efficient database storage. Codetable files are located in the `source/codetable` directory of a component. Like message files, code table files are shipped with the Social Program Management Platform and may be customized through the merge behavior.

The **ctgen** build target merges Cúram code table (`.ctx`) files and then translates the resultant merged code table file (which is stored in `/build/svr/codetable/scp` directory) into *Java* source code and SQL files which are used to return codes from the database at runtime.

The `prp.noninternedstrings` parameter indicates whether code table artefacts should be generated with strings that are not interned. The use of interned strings in *Java* avoids the creation of duplicate `java.lang.String` objects. Consequently memory usage may be reduced as there will be only one `String` object created for a string value, ir-

respective of how many references to that string value exist.

> ℹ **Note**
>
> The default value for this property is `true`. Setting `prp.noninternedstrings` to `false` means that strings will be interned. Although this may result in decreased memory usage by the final application, dependency checking will operate incorrectly when `.ctx` files are changed.

The generated *Java* code is then compiled and packed into `/build/jar/codetable.jar`.

evgen

Events provide a mechanism for loosely-coupled parts of a Cúram application to communicate information about state changes in the system. When one module in the application raises an event, one or more other modules receive notification of that event having occurred provided they are registered as listeners for that event. Event files are located in the `events` directory of a component.

The **evgen** build target merges Cúram event (`.evx`) files and then translates the resultant merged event file (which is stored in `/build/svr/events/scp` directory) into *Java* source code which can be subsequently used as constants in the application and also `.dmx` files which are used to populate the event class and event type database tables.

The generated *Java* code is then compiled and packed into `/build/jar/events.jar`.

compile.generated

The **compile.generated** target compiles any generated source code that doesn't depend on the `impl` directory. This includes the classes with the following patterns from the `build/svr/gen` directory:

```
**/struct/**/*.java
**/intf/**/*.java
<Project Package>/*.java
```

This step uses an augmented version of *Ant*'s dependency checker to minimize the build time.

3.5.2 implemented

This target completes the build and wraps the following targets:

- **compile.implemented** - this compiles all hand-crafted source code and any generated code that wasn't built during the **compile.generated** step. Again this step uses an augmented version of *Ant*'s dependency checker to minimize the build time.

compile.implemented

The **compile.implemented** step simply compiles all hand-crafted source code and any generated code that wasn't built during the **compile.generated** step. This includes the classes with the following patterns from the `build/svr/gen` directory:

```
**/base/**/*.java
**/fact/**/*.java
**/rules/loaders/*.java
**/rules/rdo/*.java
**/remote/**/*.java
```

From the `components/*/source` directory -

```
**/impl/**/*.java
**/rules/loaders/*.java
**/webservice/**/*.java
```

3.6 Extra Targets

A number of extra targets are provided which are not necessary to build a server. Some of the more useful targets are listed below:

- **clean** - Delete all the generated and compiled files to ensure all generated and compiled artefacts are removed and the next build is fresh and clean. It is useful to periodically perform clean builds because of limitations in the dependency checker provided by *Ant*.

- **encrypt** - Encrypt a plain-text database password so the encrypted password can be safely stored in a property file. None of the Cúram property files contain plain-text passwords so the passwords contained within them are automatically decrypted.

- **database** - This transforms the database independent xml files into DDL files and places the contents of these DDL files on the database. The **database** target also provides support for applying rule sets to the database (more detail on this is provided in Section 3.8, *Rules Targets*).

- **mergeshortnames** - Merges file `ShortNames.properties` from all components

- **extractdata** - This extracts the contents of all or some of the tables on the database and transforms them into database independent XML files. More detail on this target is provided in Chapter 5, *Data Manager*.

- **reloadextracteddata** - This reloads data that was extracted using the **extractdata** command back onto the database.

- **checksql** - This validates the hand-crafted SQL and test data against the actual database. If this step is not run syntactical (and semantic) mistakes in hand-crafted SQL will not be determined until run-time because of the dynamic nature of JDBC (*Java* Database Connectivity)[1]. This step operates by producing an SQLJ file and completely relies on the syntax checking provided by the particular database. The **checksql** target uses the output that is built during the **database** target. So it is a pre-requisite

to have run **database** target before running **checksql**. Any errors that are discovered while running the **checksql** target are logged to the console and to a timestamped log file in the buildlogs directory. More detail on this target is provided in Chapter 6, *SQL Checker*.

- **deprecationreport** - The command-line *Java* compiler deprecation warnings have been extended to apply to certain Cúram builds and validations. This helps to quickly pinpoint where custom dependencies exist on deprecated out-of-the-box artefacts. This target combines all the Cúram builds and validations that support deprecation warnings. As such, the build output from this target provides a comprehensive overview of all deprecation warnings for all supported builds (server and client builds, workflow validations, rules validations, etc). Please note that this target starts with a **clean** (as the *Java* compiler does not produce warnings for incremental builds). See Chapter 12, *Deprecation* for more information.

- **foreignkeycheck** - In a production environment it is not desirable to enable foreign keys on the database because of the result performance degradation. As a result it is possible for referential integrity to be violated as a result of program bugs or manual intervention by a Database Administrator. This target validates that the Referential Integrity has not been violated. It performs this task by loading the generated foreign key constraints for the application and verifying that for each child record of each foreign key the referenced parent key exists. The key values of any missing parent key records are reported.

- **test** - Execute the tests associated with the application.

  - If *Clover* is available a code coverage report can also be generated. More details on the usage of *Clover* are available in Section 3.7, *Clover Targets*.

  - The `JUnit` forkmode controls the number of *Java* Virtual Machines that gets created if you want to fork some tests; and it can be set dynamically by specifying `junit.fork.mode` property, while executing the **test** target.

    For Example:

    ```
    build test -Djunit.fork.mode=once
    ```

    Possible values for this property are:

    `perTest` - creates only a single Jav*Java* VM for all tests.

    `perBatch` - creates a *Java* VM for each nested batch test and one collecting all nested tests.

    `once` - creates only a single *Java* VM for all tests.

    Default value of `perTest` is used if `junit.fork.mode` property is not set.

- It is possible to exclude or include set of tests while running the `test` target. To Exclude/Include tests, copy the `Exclude-Tests.txt` or `IncludeTests.txt` file located in the `CuramSDEJ\util\` directory. This new file can then be modified to add the tests that you want to exclude or include and can be reference using the property override.

  For Example:

  ```
  build test
  -Dexclude.test.file=<PATH_TO_THE_FILE>\ExcludeTests.txt
  ```

  ```
  build test
  -Dinclude.test.file=<PATH_TO_THE_FILE>\IncludeTests.txt
  ```

- **configtest** - Examine the current environment to ensure that the various environment settings and property files have been established correctly. This tool attempts to diagnose any problems in the environment which would be an impact. It checks the validity of the:

  - versions of third party tools including *Java® SE Runtime Environment* (*JRE*), *Ant*, application server and database.

  - `Bootstrap.properties` including properties `curam.db.name` or `curam.db.oracle.servicename`, `curam.environment.bindings.location`, `curam.db.username`, `curam.db.password` and `curam.db.type`

  - database connectivity by attempting to connect to the database described by properties in `Bootstrap.properties` and ensures it is a valid database.

  - database configuration e.g. *DB2* buffer pools and tablespaces, *Oracle* privileges for the Cúram user

  - application server variables: `WAS_HOME` and `WLS_HOME` dependencies are also checked i.e. if using *WebSphere* the IBM® *JDK* and IBM® *Java EE* must be used.

  - *Ant* variables i.e. `ANT_HOME` and `ANT_OPTS`

  - server and client environment variables

- **configreport** - Create a `config_report.zip` file containing all the relevant settings and software versions on the machine. This can be used if remote support is required.

- **javadoc** - Produce the *Java* Documentation (JavaDoc) from the application. To produce useful JavaDoc, comments must have been placed in the model as well as in the code.

- **apijavadoc** - Generates the javadoc for black/grey box components, this is based on the javadoc.properties files.

- **release** - Gathers all the files together that are necessary to run Cúram on another machine in the `<SERVER_DIR>/release` directory. This target is used when building for a target platform (e.g. building on *Windows* for deployment on *IBM® z/OS®*) or moving the application between machines. On moving the release directory to another machine a `Bootstrap.properties` and `AppServer.properties` property files must be placed in a `release/project/properties` directory and the following environment variables must be set: `SERVER_DIR` must point at the release directory, `SERVER_MODEL_NAME` must be set to the name of the application model, and `CURAMSDEJ` must be set to the location of the *SDEJ* before any of the scripts can be used. The `SERVER_COMPONENT_ORDER` environment variable must be set on your target environment where you plan to work with the resulting release directory, and this value must be the same as the value used in your source environment. The files that are copied are:

  - *Ant* Build files;

  - Project jars;

  - DDL files;

  - SQL files;

  - Code Tables files;

  - Batch Launcher;

  - Data Manager;

  - Application EAR files.

  - XML Server files.

- **insertproperties** - Merges all the properties (.prx) files defined under the `properties` directory for each of the application's components, and inserts them into the database. See Section 4.2, *Application Properties* for more details.

- **extractproperties** - Extracts the properties from the database, and stores them into a database independent prx file. The generated prx file is stored at `<SERVER_DIR>/build/propertiesextractor/`

- **mergeuserpreferenceproperties** - Merge the user preference properties files.

- **model** - Extract the model and generate source code and other artefacts from the XML representation of a Cúram application. The **model** target combines the **modelext** and **modelgen** targets.

- **runbatch** - Runs the Batch Launcher. For more information refer to the

Cúram Batch Processing Guide.

- **runstatistics** - Runs statistics for the database. For more information refer to Section 5.8, *Statistics*.

- **supplement** - Compiles and jars all the *Java* files contained within any supplementary directory specified by the `-Dsupple-ment=<DIRECTORY_NAME>` parameter. A `<DIRECTORY_NAME>.jar` file will be created and stored in the `<SERVER_DIR>/build/jar/` directory.

- **police.access.restrictions** - Provides a report of accesses to restricted APIs within the Cúram application. The APIs that are restricted are marked by annotations within the Javadoc and indicate areas that should not be accessed by custom code. This policing tool highlights any code that accesses restricted APIs and out-of-the-box code containing a restricted annotation. During development these restrictions are further backed by the non-delivery of sample *Java* files, *Eclipse* access restrictions and that there is no JavaDoc available.

3.7 Clover Targets

Clover is a code coverage tool that can easily be integrated into the Cúram build environment. A number of *Ant* targets are provided to aid in the integration of *Clover*. For these targets to work correctly the `clover.jar` and `clover.license` files must be obtained and installed in the `<ANT_HOME>/lib` directory. More information on obtaining and using *Clover* can be found at http://www.atlassian.com/software/clover/overview.

- **clover.server** - This is the equivalent of the **server** target and also includes instrumenting the compiled `.java` files with the necessary *Clover* information.

- **clover.supplement** - This is the equivalent of the **supplement** target and also includes instrumenting the compiled `.java` files with the necessary *Clover* information.

- **clover.report.html** - This target will generate a html report detailing code coverage. The report is generated into the `<SERVER_DIR>/clover/clover_html` folder.

- **clover.report.viewer** - This target will launch the *Clover* viewer with details of the code coverage.

3.8 Rules Targets

The *Cúram Rules Codification Guide* and the *Cúram Rules Editor Guide* provide an introduction to the support for rules in Cúram. A rule set is the fundamental structure which describes the *rules* within a Cúram application. It is the database that is the system of record for rule sets. This allows the

rule sets to be changed at run-time via an administration client. However, support is also provided for representing rule sets as `.xml` files. These `.xml` files can be used for source control management. To allow for the synchronization between these `.xml` files and the database a number of extra targets have been introduced:

> ℹ **Representing Rulesets as XML Files**
>
> Support for ruleset import and export is only there to allow source control management and to exchange rulesets between machines. Direct editing of the ruleset XML files is not supported in any way.

- **listrulesets** - Produce a listing of the names and identifiers of the rulesets that are present on the database.

- **exportruleset** - This target exports a ruleset definition (.xml file) from database to the file system. This command takes two parameters - rulesetid and component. Exported ruleset will be saved as [specified rulesetid].xml in `<SERVER_DIR>/components/[specified component]/rulesets` folder.

  `rulesetid` - Identifier of the ruleset that is to be exported from the database.

  `component` - Name of the component to which the rule set has to be exported (copied).

  For example:

  ```
  build exportruleset
  -Drulesetid=PRODUCT_1
  -Dcomponent=custom
  ```

  Where 'PRODUCT_1' denotes the identifier of the ruleset that is to be exported from the database and 'core' denotes the name of the component to which the rule set has to be exported (copied).

- **importruleset** - This target imports a ruleset definition (.xml file) from a file system to the database. It validates the rule set ID for uniqueness before importing the rule set, it does this by searching for existing IDs in the `SERVER_DIR/components/../rulesets` directories. This command takes two parameters- ruleset.file and overwrite.

  `ruleset.file` - This parameter denotes the path of the ruleset that is to be placed on the database.

  `overwrite (Optional)` - This is an optional flag with the default value as `false`, indicating whether the database should be overwritten if the ruleset already exists.

  For example:

  ```
  build importruleset
  -Druleset.file=
    <SERVER_DIR>/components/core/rulesets/PRODUCT_1.xml
  -Doverwrite=true
  ```

Where
`<SERVER_DIR>/components/core/rulesets/PRODUCT_1.`
`xml` denotes the path of the ruleset definition file and #true# denotes the
flag to overwrite the database, if ruleset already exists.

- **validateallrulesets** - Validates all the rule sets in the Cúram application.
  This target has to be invoked from the `SERVER_DIR` directory, where it
  scans all the components for rule set files and validates them. For
  schema validation this target uses the rule set schema located in
  `CURAMSDEJ/lib` directory by default, unless another schema is spe-
  cified by using an optional property 'schema.file'.

  The validator ensures that the rule set ID is unique by searching for ex-
  isting IDs in the `SERVER_DIR/components/../rulesets` dir-
  ectories.

  `schema.file (Optional)` - This optional parameter specifies the
  rule set schema that has to be used for validating the rule sets.

  For example:

```
ant validateallrulesets
```
```
ant validateallrulesets
            -Dschema.file=C:/Rules/ruleset.xsd
```

- **validaterulesets** - Validates all the rule sets in the specified directory.
  The property 'rulesets.dir' has to be specified when invoking the target.
  For schema validation this target uses the rule set schema located in
  `CURAMSDEJ/lib` directory by default, unless another schema is spe-
  cified by using an optional property 'schema.file'.

  The validator ensures that the rule set ID is unique by searching for ex-
  isting IDs in the `SERVER_DIR/components/../rulesets` dir-
  ectories.

  `schema.file (Optional)` - This optional parameter specifies the
  rule set schema that has to be used for validating the rule sets.

  `rulesets.dir` - This parameter specifies the directory within which
  rule sets are to be validated.

  For example:

```
ant validaterulesets
            -Drulesets.dir=
              <SERVER_DIR>/components/core
```
```
ant validaterulesets
            -Drulesets.dir=
              <SERVER_DIR>/components/core
                -Dschema.file=C:/Rules/ruleset.xsd
```

- **validateruleset** - Validates the specified rule set. The property 'rule-
  set.file' that denotes the rule set path and file name has to be specified
  when invoking the target. For schema validation this target uses the rule
  set schema located in `CURAMSDEJ/lib` directory by default, unless

another schema is specified by using an optional property 'schema.file'.

The validator ensures that the rule set ID is unique by searching for existing IDs in the `SERVER_DIR/components/../rulesets` directories.

`schema.file (Optional)` - This optional parameter specifies the rule set schema that has to be used for validating the rule set.

`ruleset.file` - This parameter specifies the rule set path and file name.

For example:

```
ant validateruleset
          -Drulesets.file=
          <SERVER_DIR>/components/core/rulesets/PRODUCT_1.xml
```

```
ant validateruleset
          -Drulesets.file=
            <SERVER_DIR>/components/core/rulesets/PRODUCT_1.xml
            -Dschema.file=C:/Rules/ruleset.xsd
```

- **rulesfunctionsmerge** - Merge rules custom function meta-data from .xml files.

3.9 Classic IEG Targets

The *Classic Intelligent Evidence Gathering Guide* provides an introduction to scripts and question groups within Classic IEG. The database can be populated by using the Classic IEG editor to define these scripts. However, support is also provided for representing the groups and scripts as xml data. Although the files are consistent with well and fully formed xml, the file extensions are modified to denote the contents as script (`.sx`) and question group(`.gx`). These xml data files can be created and manipulated directly to allow for the synchronization between these files, and the database. A number of extra targets have been introduced to enable this:

- **importieg** - This target imports all IEG files in a specified directory to the database. This command takes two parameters - directory and overwrite.

`directory` - This parameter denotes the directory from which IEG scripts and question groups are imported.

`overwrite (Optional)` - This is an optional flag with the default value as `false`, indicating whether the database should be overwritten if an imported script or group already exists.

For example:

```
build importieg
  -Ddirectory=
     <SERVER_DIR>/components/core/ieg
  -Doverwrite=true
```

Where `<SERVER_DIR>/components/core/ieg` denotes the path to the import directory and #true# denotes the flag to overwrite the database if a file already exists.

- **importiegscript** - This target imports an IEG script from a file system to the database. This command takes two parameters - IEG file and overwrite.

  `ieg.file` - This parameter denotes the full path of the IEG script to be imported.

  `overwrite` (Optional) - This is an optional flag with the default value as `false`, indicating whether the database should be overwritten if the script already exists.

  For example:

  ```
  build importiegscript
    -Dieg.file=
       <SERVER_DIR>/components/core/ieg/PRODUCT_1.sx
    -Doverwrite=true
  ```

  Where
  `<SERVER_DIR>/components/core/ieg/PRODUCT_1.sx` denotes the path of the script definition file and 'true' denotes the flag to overwrite the database if the question script already exists.

- **importiegcomponent** - This target imports all IEG data (IEG scripts and question groups) from the `ieg` subdirectory of a specified component to the database. This command takes two parameters - component and overwrite.

  `component` - This parameter denotes the component from which to import all IEG data to the database.

  `overwrite` (Optional) - This is an optional flag with the default value as `false`, indicating whether the database should be overwritten if the IEG data already exists.

  For example:

  ```
  build importiegcomponent
    -Dcomponent=core
    -Doverwrite=true
  ```

  Where `core` denotes the path of the component and 'true' denotes the flag to overwrite the database if the question script already exists.

- **importiegsubdirs** - This target imports all IEG data (IEG scripts and question groups) from the `ieg` subdirectory of subdirectories of a specified directory to the database. This command takes two parameters - directory and overwrite. This target is used when it is required to import IEG data from multiple components.

directory - This parameter denotes the directory whose subdirectories will be searched for IEG data to import to the database.

overwrite (Optional) - This is an optional flag with the default value as false, indicating whether the database should be overwritten if the IEG data already exists.

For example:

```
build importiegsubdirs
  -Ddirectory=<SERVER_DIR>/components
  -Doverwrite=true
```

Where <SERVER_DIR>/components denotes the path of the directory and 'true' denotes the flag to overwrite the database if the IEG data already exists.

• **importquestiongroup** - This target imports an IEG question group from a file system to the database. This command takes two parameters - IEG file and overwrite.

ieg.file - This parameter denotes the full path of the IEG question group to be imported.

overwrite (Optional) - This is an optional flag with the default value as false, indicating whether the database should be overwritten if IEG import data already exists.

For example:

```
build importquestiongroup
  -Dieg.file=
    <SERVER_DIR>/components/core/ieg/PRODUCT_1.gx
  -Doverwrite=true
```

Where
<SERVER_DIR>/components/core/ieg/PRODUCT_1.gx denotes the path of the question group definition file and #true# denotes the flag to overwrite the database, if the question group already exists.

• **exportiegscript** - This target exports a script definition (.sx file) from a database to the file system. This command takes two parameters - scriptid and component. Exported scripts will be saved and named as [specified rulesetid].sx in the <SERVER_DIR>/components/[specified component]/ieg folder.

scriptid- Identifier of the script that is to be exported from the database.

component- Name of the component to which the script has to be exported (copied).

For example:

```
build exportiegscript
  -Dscriptid=PRODUCT_1
  -Dcomponent=core
```

Where 'PRODUCT_1' denotes the identifier of the script that is to be exported from the database and 'core' denotes the name of the component to which the script has to be exported (copied).

- **exportiegscripttodir** - This target exports a script definition (.sx file) from a database to the file system. This command takes two parameters - scriptid and exportdirectory. Exported scripts will be saved and named as [specified rulesetid].sx in the specified export directory.

  `scriptid`- Identifier of the script that is to be exported from the database.

  `exportdirectory`- Full path of the directory to which the script has to be exported (copied).

  For example:

```
build exportiegscripttodir
  -Dscriptid=PRODUCT_1
  -Dexportdirectory=C:/exportedscripts
```

  Where 'PRODUCT_1' denotes the identifier of the script that is to be exported from the database and 'C:/exportedscripts' denotes path to the directory to which the script has to be exported (copied).

- **exportfulliegscript** - This target exports a specific script definition (.sx file) and its associated group definitions(.gx files) from the database to the file system. If any of the files exported are read only, a warning will be reported and the file will not be overwritten. The exportfulliegscript command takes two parameters - scriptid and component. The exported script is named as [specified scriptid].sx and the associated question groups are saved and named as [associated questiongroupid].gx in the `<SERVER_DIR>/components/[specified compon-ent]/ieg` folder.

  `scriptid`- Identifier of the script that is to be exported from the database.

  `component`- Name of the component to which the script has to be exported (copied).

  For example:

```
build exportiegscript
  -Dscriptid=PRODUCT_1
  -Dcomponent=core
```

  Where 'PRODUCT_1' denotes the identifier of the script that is to be exported from the database and 'core' denotes the name of the component to which the script and its associated question groups has to be exported

(copied).

- **exportfulliegscripttodir** - This target exports a specific script definition (.sx file) and its associated group definitions(.gx files) from the database to the file system. If any of the files exported are read only, a warning will be reported and the file will not be overwritten. The exportfulliegscripttodir command takes two parameters - scriptid and exportdirectory. The exported script is named as [specified scriptid].sx and the associated question groups are saved and named as [associated questiongroupid].gx in the specified export directory.

  `scriptid`- Identifier of the script that is to be exported from the database.

  `exportdirectory`- Full path of the directory to which the script has to be exported (copied).

  For example:

  ```
  build exportfulliegscripttodir
    -Dscriptid=PRODUCT_1
    -Dexportdirectory=C:/exportedscripts
  ```

  Where 'PRODUCT_1' denotes the identifier of the script that is to be exported from the database and 'C:/exportedscripts' denotes the path to the directory to which the script and its associated question groups has to be exported (copied).

- **exportquestiongroup** - This target exports a question group definition (.gx file) from database to the file system. This command takes two parameters - groupid and component. Exported question groups will be saved as [specified rulesetid].gx in `<SERVER_DIR>/components/[specified component]/ieg` folder.

  `groupid`- Identifier of the question group that is to be exported from the database.

  `component`- Name of the component to which the rule set has to be exported (copied).

  For example:

  ```
  build exportquestiongroup
    -Dgroupid=PRODUCT_1
    -Dcomponent=core
  ```

  Where 'PRODUCT_1' denotes the identifier of the question group that is to be exported from the database and 'core' denotes the name of the component to which the question group has to be exported (copied).

- **listiegscripts** - Produces a list of all the IEG scripts available in the database.

- **listquestiongroups** - Produces a list of all the question groups available

in the database.

- **migrateiegscript** - Migrate a Classic IEG script definition to an IEG2 script definition (use -Dscriptfilename= -Dinputdir= -Doutputdir=).

  `scriptfilename` - denotes the name of the file to import.

  `inputdir` - denotes the directory to import from.

  `outputdir` - denotes the directory to write the migrated script to.

- **validatealliegscripts** - Validates the IEG scripts.

3.10 IEG2 Targets

- **validateieg2scripts** - Validates the IEG2 scripts in the specified directory. Requires IntelligentEvidenceGathering component to run.

3.11 Application Configuration Import and Export Targets

The application configuration information for the Cúram web client is stored as a series of XML and properties files in the server source directory. It is merged and loaded into the database at build time from where it is read by the client tier at run time.

The rules for merging are as follows:

- Files in the `clientapps` directory take precedence over files in the `tab` directory, regardless of component order. E.g: if a file named `CaseHome.nav` exists in the `clientapps` directory of any component of the application, then any files named `CaseHome.nav` which exist in the `tab` directory of any component are ignored.

- Files in the `clientapps` directory are selected (not merged) based on the component order. E.g: if a file name `CaseHome.nav` exists in the `clientapps` directory of components Custom1 and Custom2, and Custom1 is ahead of Custom2 in the component order, then the version of `CaseHome.nav` from Custom1 is used and the version from Custom2 is ignored.

- Files in the `tab` directory are merged according to the component order - provided that a corresponding file in a `clientapps` directory does not exist. E.g: if a file named `SearchTab.nav` exists in the `tab` directory of components CustomA and CustomB, but not in the `clientapps` directory of any component, then the contents of the two files are merged together.

  > **Note**
  >
  > Note that only OOTB Cúram components may use the `tab` dir-

ectory to store application configuration files; this directory may not be used by custom components. Custom components may use only the `clientapps` directory for application configuration files.

One target controls the import and export of application configuration to and from the database:

inserttabconfiguration

Merges application configuration files from disk and inserts the data into the database. The default action of this target is to insert the application configuration data onto the database but it can also be used to:

- Merge the application configuration files and write the merged files to a directory on disk.

  If property `dir.tab.merge` is set then it denotes a directory into which the application configuration files from the various components of your application will be merged. In this mode, nothing is written to the database. E.g: **build inserttabconfiguration -Ddir.tab.merge=C:/EJBServer/tabExtract**

- Extract the application configuration data from the database and write it to a directory on disk.

  If property `dir.tab.extract` is set then it denotes a directory into which the application configuration data from the database will be extracted. In this mode the application configuration data is read from the database and nothing is written to the database. E.g: **build inserttabconfiguration -Ddir.tab.extract=C:/EJBServer/tabExtract**

3.12 Workflow Targets

The *Cúram Workflow Reference Guide* provides an introduction to the support for workflow in Cúram. A workflow process definition is the fundamental structure which describes the *workflow* process within a Cúram application. Workflow process definitions are stored on the database, but can also be represented as `.xml` files and loaded onto the database as needed. A number of targets exist to allow for the validation of workflow process definition `.xml` files:

> **Prerequisites for validating workflow process definition files**
>
> Workflow process definitions can make reference to rule sets (see *Cúram Rules Codification Guide*) and Cúram events (See Chapter 18, *Events and Event Handlers*) in the process `xml` files. Therefore, all rule sets and events that are referenced in workflow process definitions being validated must already be loaded onto the database before the associated workflow process definition files can be validated using the targets outlined below.

- **validateworkflows** - supports validation of the workflow process definition files in the specified directory. The property 'workflow.dir' has to be specified when invoking the target.

  `workflow.dir` - This parameter denotes the directory within which workflow process definition files are to be validated.

  `validate.schema.only` - This optional parameter, if set to true, only performs schema validation on the workflow `xml` files and bypasses the full semantic validation that would otherwise be performed.

  For example:

  ```
  ant validateworkflows
     -Dworkflow.dir=
        <SERVER_DIR>/path to workflow directory
  ```

- **validateallworkflows** - performs validation of all workflow process definitions files in the Cúram application.

  `validate.schema.only` - This optional parameter, if set to true, only performs schema validation on the workflow `xml` files and bypasses the full semantic validation that would otherwise be performed.

  For example:

  ```
  ant validateallworkflows
  ```

- **validateworkflow** - supports validation of the specified workflow process definition file. The property 'workflow.file' has to be specified when invoking this target.

  `workflow.file` - This parameter denotes the full path to the workflow process definition file that is to be validated.

  `validate.schema.only` - This optional parameter, if set to true, only performs schema validation on the workflow `xml` file and bypasses the full semantic validation that would otherwise be performed.

  For example:

  ```
  ant validateworkflow
     -Dworkflow.file=
        <SERVER_DIR>/path to workflow file to be validated
  ```

- **importworkflow** - Import a workflow process definition (use -Dworkflow.file= -Doverwrite=).

  `workflow.file` - This parameter denotes the full path to the workflow process definition file that is to be imported.

  `overwrite (Optional)` - This is an optional flag with the default value as `false`, indicating whether the database should be overwritten if the workflow process definition already exists.

- **importworkflows** - Import the workflow definitions in the specified directory (use -Dworkflow.dir= -Doverwrite=).

workflow.dir - This parameter denotes the directory from which the workflow definitions should be imported.

overwrite (Optional) - This is an optional flag with the default value as false, indicating whether the database should be overwritten if the workflow process definitions already exist.

- **listworkflows** - List all process definitions available in the database.

3.13 Deployment Targets

A number of extra targets exist which allow an application to be deployed on an application server. These commands are fully described in the *Cúram Deployment Guide*[2], but a summary is provided here.

- **weblogicEAR** - Produce an .ear file that can be deployed on *WebLogic*.

- **websphereEAR** - Produce an .ear file that can be deployed on *WebSphere Application Server*.

- **weblogicHelpEAR** - Produce an Online Help application Curam-Help.ear file that can be deployed on *WebLogic*.

- **websphereHelpEAR** - Produce an Online Help application Curam-Help.ear file that can be deployed on *WebSphere Application Server*.

- **weblogicWebServices -** Produce an .ear file that can be deployed on *WebLogic* that supports Web Services invocation.

- **websphereWebServices -** Produce an .ear file that can be deployed on *WebSphere* that supports Web Services invocation.

- **weblogicEARGSS** - Build GSS ear for *WebLogic*

- **websphereEARGSS** - Build GSS ear for *WebSphere Application Server*

- **configure** - Automatically configures the application server.

- **installapp** - Installs and starts a specified EJB application. (Note: the EAR file (Curam.ear) containing the server module must be deployed before installing any other (client-only) EAR files.)

- **precompilejsp** - Precompiles all JSPs in the specified .ear file.

- **prepare.application.data** - Must be run after the **database** target is run and before starting the application server for the first time. Failing to run this sequence will likely result in transaction timeouts during first login and a failure to initialize and access the application. Whenever the **database** target is rerun (e.g. in a development environment) this target must also be rerun.

- **startserver** - Starts an application server.

- **restartserver** - Restarts an application server.

- **stopserver** - Stops an application server.

- **uninstallapp** - Stops and uninstalls the specified EJB application.

3.14 Extending the Build

This section describes how *Ant* can be used to introduce new targets, enhance existing targets or override OOTB build targets.

This is achieved by creating a script hierarchy using *Ant*'s **import** task and can be seen in the OOTB application. Examples include the `build.xml` files found in the `webclient` and `EJBServer` directories that extend, through an import, the `build.xml` files from the `CuramCDEJ` and `CuramSDEJ` directories respectively.

The delivered `build.bat or .sh` files invoke *Ant* against the webclient or `EJBServer build.xml`. This allows for these `build.xml` files to introduce new targets not available in the scripts delivered in the *CDEJ* and *SDEJ*. It also allows these targets to be enhanced as required due to the principal of the import task, which is that `"If a target in the main file is also present in at least one of the imported files, the one from the main file takes precedence"`.

3.14.1 Introducing a new script

The following section details the steps to create a new top level script which can be used to introduce new targets, enhance existing targets or override OOTB build targets.

Two Environment variables `CDEJ_BUILDFILE` and `SDEJ_BUILDFILE` are used to control the script that is invoked by the `build.bat or .sh` files. A new script can be invoked by setting the appropriate environment variable. For example:

Introducing a new server script:

```
SDEJ_BUILDFILE=%SERVER_DIR%/components/custom/scripts/build.xml
```

This script must then import it's parent in the hierarchy `EJBServer\build.xml`, for example:

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

  <!--  Relative path to EJBServer\build.xml  -->
  <import file="./../../../../build.xml"/>

</project>
```

New targets can then be added to the script as required. These targets can also utilize existing targets or properties in the inherited script hierarchy.

To enhance or override an existing target the same target name is chosen as that which is being enhanced or overriden. When enhancing a target, the existing target is then either added as a dependency of the new target or invoked during a point in the new target. The previous target's name used is formed from the `project name` of the script where the target being enhanced exists. For example:

Enhancing the **database** target, where the `project name` of the SDEJ script containing the **database** target is **app_database**.

Before target usage:

```
<target name="database" >
  <!-- Some further processing before the SDEJ database target -->
  ...
  <antcall target="app_database.database"/>
</target>
```

After target usage:

```
<target name="database" depends="app_database.database">
  <!-- Some further processing after the SDEJ database target -->
  ...
</target>
```

Example 3.2 Before/After Target usage

> **ℹ** Ant Target API
>
> Only targets that are documented i.e. those visible through the **-projecthelp** for a script should be enhanced, overriden or invoked. Other targets are considered internal are subject to change without notice.

3.15 Overridden Targets

Some targets in the SDEJ are overridden by application build scripts. Such targets appear in the report produced by the **-projecthelp** command qualified by the SDEJ sub project name such as **app_auxiliary**, **serverbuild**, etc. Only the unqualified version of these targets should be used, otherwise the target may not work correctly. E.g. always use **weblogicEAR** instead of **serverbuild.weblogicEAR**.

This applies to the following targets:

- **app_auxiliary.ctgen**

- **app_auxiliary.msggen**

- **app_runtimewas.configure**

- **serverbuild.clean**

- **serverbuild.generated**

- **serverbuild.implemented**

- **serverbuild.model**

- **serverbuild.release**

- **serverbuild.weblogicEAR**

- **serverbuild.websphereEAR**

3.16 Application Targets

This section lists targets which are available in the OOTB application and which are displayed when the **-projecthelp** command is given.

3.16.1 BI App

- **biapp.BIRTViewerEARs** - Builds deployable EAR files for *WebSphere* and *WebLogic*

- **biapp.configure.birtviewer** - Configures the Cúram *Business Intelligence and Reporting Tools* (*BIRT*) Viewer application (use -Dserver.name= -Dear.file= -Dapplication.name=)

  `server.name` - The name of the server to deploy the application onto.

  `application.name` - The name of the *BIRT* Viewer application.

- **biapp.release** - Copies *BIRT* build files required to run biapp.configure.birtviewer (post install step for *WebSphere*)

3.16.2 CREOLE

- **creole.check.initial.database** - Checks the structure of rule set XML data uploaded from DMX files and runs lax validation.

- **creole.compile.test.classes** - Compiles the test classes generated from the CREOLE rule sets.

- **creole.consolidate.resource** - Consolidates together resource bundles for CREOLE rule sets.

- **creole.consolidate.rulesets** - Inlines any included CREOLE rule sets and consolidates the rule sets into one build directory.

- **creole.copyresourceto.cls** - Copies resource files for CREOLE rule sets into the build\svr\cls directory.

- **creole.generate.catalog** - Generates an XML catalog file for CREOLE

rule sets.

- **creole.generate.ruledoc** - Generates rule documentation for all CREOLE rule sets.

- **creole.generate.schema** - Generates an XML schema file for CREOLE rule sets.

- **creole.generate.test.classes** - Generates test classes from the CREOLE rule sets.

- **creole.report.coverage** - Reports CREOLE rule set coverage information gathered from CREOLE rule executions.

- **creole.report.unused.attributes** - Reports CREOLE rule attributes which are not used directly by any other rule attributes.

- **creole.upload.rulesets** - Uploads new CREOLE rule sets and/or changes to existing CREOLE rule sets to the database.

- **creole.validate.rulesets** - Performs full validation of all CREOLE rule sets.

3.16.3 Evidence Generation

- **egtools.assign.resourceID** - Allocate resourceID values for the Create Wizard AppResource.dmx.

- **egtools.clean** - Calls on the EG Tool to delete all generated components.

- **egtools.client.clean** - Calls on the EG Tool to delete all generated client evidence screens on the product.

- **egtools.client.generate** - Generate target for client evidence generation.

- **egtools.generate** - Main generate target for evidence generator. Generates all evidence components.

- **egtools.server.clean** - Calls on the EG Tool to delete all generated components on the server.

- **egtools.server.generate** - Generate target for server evidence generation.

- **egtools.wizard.dmx** - Generate target for creation of AppResource.dmx for Create Wizard pages.

- **post.modelgen** - Calls on the EG Tool to perform any steps required after the modelgen.

- **add.rootnode.to.appresource.dmx** - APPRESOURCE.dmx gets appended to by each product's evidence generation. This adds the root node 'table'.

- **add.rootnode.to.initialappresource.dmx** - INITIALAPPRE-

SOURCE.dmx gets appended to by each product's evidence generation. This adds a root node to make a valid xml file.

- **add.rootnode.to.products.xml** - Product.xml gets appended to by each product's evidence generation. This adds the root node 'products'.

- **build.all.component.dirs** - Builds all components.

- **build.all.evidence.dirs** - Builds all evidence directories.

- **build.evidencebroker.resources** - Builds the evidencebroker resources for domains and labels.

- **call.egtools.transformer** - Calls on the XSLT transformer.

- **generate.resolve.scripts** - Calls any XSLT transformations that require the cross products summary defined in Products.xml.

- **makedir** - Creates directory structure for an evidenceEntities.xml file in the EJBServer/build folder if none exists. Should only be necessary if an appbuild clean has been performed.

Notes

[1]JDBC (*Java* Database Connectivity) is part of the *Java* Development Kit which defines an application programming interface for *Java* for standard SQL access to databases from *Java* programs.

[2]For your particular application server, i.e. *WebSphere* and *WebLogic*. The deployment guides are named *Cúram Deployment Guide for WebSphere Application Server*, *Cúram Deployment Guide for WebSphere Application Server on z/OS*, and *Cúram Deployment Guide for WebLogic Server*.

Chapter 4

Cúram Configuration Settings

4.1 Overview

This chapter details the environment variables that can be set in your *IBM Cúram Social Program Management* environment.

4.2 Application Properties

This section describes the property files associated with developing or running a Cúram application.

4.2.1 Application.prx

The `Application.prx` contains the properties used when running a Cúram application. The properties contained in this file are loaded to the database during the **build database** target and at runtime are cached from the database for use by the Application. An `Application.prx` can be loaded separately via the **build insertproperties** target.

The properties defined in `Application.prx` can be *dynamic* or *static*. *Dynamic* properties will have effect immediately if changed and published using the administration interface during runtime. Modifying *static* properties will have no effect until a restart of the server is performed.

```
<property name="curam.trace" dynamic="true">
  <type>STRING</type>
  <value>trace_ultra_verbose</value>
  <default-value>trace_ultra_verbose</default-value>
  <category>CODETABLE</category>
  <locales>
    <locale language="en" country="US">
      <display-name>Trace Property</display-name>
      <description>Details of the Trace Property</description>
    </locale>
  </locales>
```

```
    </property>
```

Example 4.1 PRX Entry

The file is organized as follows:

Property Element

> There is a property element for each property used.

> **Name Attribute**

>> Attribute specifying the name of the property.

> **Dynamic Attribute**

>> Indicator as to whether a change to the property value will require an Application restart.

> **Type Element**

>> Refers to a code entry on the codetable `DomainType`.

> **Value Element**

>> The property value.

> **Default-Value Element**

>> The default value of a property used when properties are reset.

> **Category Element**

>> Refers to a code entry on the codetable `EnvPropertyCategory`.

> **Locales Element**

>> Contains one or more locale specific elements for the display name and description.

>> **Language Attribute**

>>> Language Code for this locale specific entry.

>> **Country Attribute**

>>> (Optional) Country Code for this locale specific entry.

>> **Display Name Element**

>>> Locale specific display name for the property.

>> **Description Element**

>>> Locale specific entry for the property.

Merging an Application prx File

Prx files are located in the `/properties` directory of a component and the root `/project/properties` directory. The Social Program Management Platform is shipped with a set of prx files. These may be overridden by placing new prx files in the `SERVER_DIR/components/<custom>/properties` directory, where

<custom> is any new directory created under components that conforms to the same directory structure as components/core. This mechanism avoids the need to make changes directly to the out-of-the-box application, which would complicate later upgrades.

This override process involves merging all prx files according to a precedence order. The order is based on the SERVER_COMPONENT_ORDER environment variable. This environment variable contains a comma-separated list of component names: the left most has the highest priority, and the right most the lowest.

```
SERVER_COMPONENT_ORDER=custom,Appeal,ISProduct,sample
```

Example 4.2 SERVER_COMPONENT_ORDER example

The order shows that the precedence of Appeal is higher than that of the sample component. The core component always has the lowest priority and as such does not need to be specified. Any components which are not specified are placed alphabetically above core and below those which are specified.

> ℹ️ **Note**
>
> After changing the component precedence order in SERVER_COMPONENT_ORDER it is necessary to preform a re-insert of the merged properties. This is done by invoking **build insertproperties**.

When merging prx files, the components listed in the SERVER_COMPONENT_ORDER are taken in order of highest to lowest priority. In the above example the Application.prx file from the sample component is merged with the Application.prx located in the core component. The Application.prx from ISProduct is then merged into the intermediate results and the merge process continues until the Application.prx in the custom component is merged.

Rules of PRX Merges

PRX files are merged based on precedence order. As described above there is always a more important main/source Application.prx file, and a file which is being merged into it. The second file is called the merge file in the following sections.

An Application.prx files can be customized by:

- Adding a property providing mandatory property values.

- Overriding an existing properties description.

- Overriding an existing properties display name.

- Override an existing properties value or default value.

- Adding a new locale to provide a new display name and description for

that locale.

- Removing a property by setting the property tag `removed` to be `true`.

An `Application.prx` files cannot be customized by:

- Changing an existing property name.

- Changing an existing properties type.

- Changing an existing properties category.

- Changing the static or dynamic setting of a property .

Duplicate property nodes will always be overwritten by the `Application.prx` file in the component with the highest precedence order. The main `Application.prx` example file below and the merge `Application.prx` file below illustrate these rules:

```
<property name="curam.trace" dynamic="true">
  <type>STRING</type>
  <value>trace_ultra_verbose</value>
  <default-value>trace_ultra_verbose</default-value>
  <category>CODETABLE</category>
  <locales>
    <locale language="en" country="US">
      <display-name>Trace Property</display-name>
      <description>Details of the Trace Property</description>
    </locale>
  </locales>
</property>
```

Example 4.3 Sample main Application.prx file

```
<property name="curam.trace" dynamic="true">
  <type>STRING</type>
  <value>trace_off</value>
  <default-value>trace_off</default-value>
  <category>CODETABLE</category>
  <locales>
    <locale language="en" country="GB">
      <display-name>New Trace Display Name</display-name>
      <description>New Description</description>
    </locale>
  </locales>
</property>
<property name="property2" dynamic="true">
  <type>STRING</type>
  <value>value</value>
  <default-value>default</default-value>
  <category>CODETABLE</category>
  <locales>
    <locale language="en" country="GB">
      <display-name>Display Name</display-name>
      <description>Description</description>
    </locale>
  </locales>
</property>
```

Example 4.4 Sample merge Application.prx file

As a result of the merge process the new `Application.prx` produced would be:

```
<property name="curam.trace" dynamic="true">
  <type>STRING</type>
  <value>trace_off</value>
  <default-value>trace_off</default-value>
  <category>CODETABLE</category>
  <locales>
    <locale language="en" country="US">
      <display-name>Trace Property</display-name>
      <description>Details of the Trace Property</description>
    </locale>
    <locale language="en" country="GB">
      <display-name>New Trace Display Name</display-name>
      <description>New Description</description>
    </locale>
  </locales>
</property>
<property name="property2" dynamic="true">
  <type>STRING</type>
  <value>value</value>
  <default-value>default</default-value>
  <category>CODETABLE</category>
  <locales>
    <locale language="en" country="GB">
      <display-name>Display Name</display-name>
      <description>Description</description>
    </locale>
  </locales>
</property>
```

Example 4.5 Resulting Application.prx File

4.2.2 Bootstrap.properties

The `Bootstrap.properties` file mainly contains the minimum set of properties necessary for obtaining a connection to the database. These properties will generally have no effect if set in the `Application.prx` file and are only picked up directly from the `Bootstrap.properties` file.

The `Bootstrap.properties` file may also contain properties that can be defined in `Application.prx` file. If such a property is defined in the `Bootstrap.properties` file and is a dynamic property, it can be overridden by setting it on database using the administration interface.

> **Note**
>
> Properties defined in the following are cached: `Application.prx`, `Bootstrap.properties` and *Java* System properties at runtime. Properties defined in `Application.prx` are loaded into the database and can be updated at runtime using the administration interface. A publish is required to rebuild the property cache and allow the changes to take effect.
>
> The property cache loads its contents with the following priority:
>
> 1. *Java* System properties,

2. `Application.prx,`

3. `Bootstrap.properties;`

For example, if a property is set in the *Java* System properties (either via the Application Server or using `java.lang.System.setProperty()`) and also in `Applic-curam.util.resources.Configuration.g ation.prxetProperty()`, the value of the property defined in the *Java* System properties will always be returned when using the `Application.prx` and `Bootstrap.properties`, the value of the property in `Application.prx` is what will take effect.

```
# Tnameserv Port
curam.environment.tnameserv.port=900
curam.environment.bindings.location=C:/Bindings

curam.db.username=db2admin
curam.db.password=wWw5UTMnFOe1SeCBEQy/Zg==
curam.db.type=DB2
curam.db.name=CURAM
curam.db.serverport=50000
curam.db.servername=localhost

# property to specify Oracle service name.
curam.db.oracle.servicename=orcl.<host_name>

# Properties specific to H2
# Mode remote|embedded
curam.db.h2.mode=embedded
# For remote mode also specify:
curam.db.serverport=9092
curam.db.servername=localhost
# Lock Time Out in ms. Default is 1000, i.e. 1 second. (Optional)
curam.db.h2.locktimeout=20000
# Property to disable MVCC. Default: true. (Optional)
curam.db.h2.mvcc=true
```

Example 4.6 Bootstrap.properties

An automatically generated version of `Bootstrap.properties` is packed in the Enterprise Archive (EAR) when building the EAR file. This file chooses it's properties from the default `Bootstrap.properties` and is extended with extra properties relating to the Application Server being used.

```
curam.db.type=DB2
curam.environment.as.vendor=IBM
```

Example 4.7 Bootstrap.properties in an EAR file

ℹ️ Note

The EAR file cannot be built for *H2* database. [1]

4.3 Support for Multiple Time Zones

To enable multiple time zone support, the time zone ID must be specified for each user in the user preferences.

Only DateTimes are processed and displayed in the user's preferred time zone. Date only and Time only fields are not affected and for these fields it is the responsibility of the business logic to ensure that the time zone is not relevant. If the time zone is relevant then a DateTime field should be used. An example of a date where the time zone is not relevant is someone's date of birth; it doesn't vary no matter what time zone that person was born. An example of a date where the time zone is relevant is the current date; this will be different for two user's working either side of the international date line, in this case a DateTime must be used.

The server's time zone is basically the underlying operating system's configured time zone, however the server stores date/times in a time zone independent manner, i.e the number of milliseconds since 1/1/1970 00:00 GMT (also known as the epoch). It is the responsibility of the web tier to convert all DateTimes passed to it from the server into the user's preferred time zone and also to convert all DateTimes to be passed back to the server into milliseconds since the epoch.

The preferred time zone for each user is configured based on the time zone ID specified in the user preferences for the particular user. The time zone ID must conform to one of the time zones returned from the *Java* method `java.util.TimeZone.getAvailableIDs()`.

Some of the *Java* supported time zones returned by `java.util.TimeZone.getAvailableIDs()` method are as listed below:

- *GMT+x ,where x can take value from 1 to 12.*

- *GMT-x ,where x can take value from 1 to 12.*

- *America/Chicago*

- *America/Mexico_City*

- *America/Indiana/Indianapolis*

- *America/New_York*

- *America/Los_Angeles*

- *Australia/Canberra*

- *Australia/North*

- *Australia/South*

- *Australia/West*

- *Australia/Adelaide*

- *Australia/Melbourne*

- *Australia/Brisbane*

- *Africa/Casablanca*

- *Africa/Johannesburg*

- *Brazil/West*

- *Canada/Pacific*

- *Canada/Saskatchewan*

- *Canada/Eastern*

- *Canada/Atlantic*

- *Canada/Central*

- *Canada/Eastern*

- *Europe/London*

- *Europe/Dublin*

- *Europe/London*

- *Europe/Paris*

- *Europe/Vatican*

- *Europe/Moscow*

- *Europe/Amsterdam*

- *Indian/Chagos*

- *Indian/Cocos*

- *NZ*

- *Pacific/Auckland*

For information on server time zone configuration consult the Time Zone Configuration chapter in the *Cúram Deployment Guide* for the appropriate application server.

4.4 Dates and date/times in Cúram

This section describes the behavior of dates and date/times in Cúram.

Take a look at these examples:

- The server is in time zone "GMT". A user is in time zone "GMT -01". At 15:00 GMT the user registers a new person, and the server-side processing timestamps a resulting database record with the time 15:00. Twenty seconds later the user performs a query and sees the timestamp displayed in the client user interface as 14:00. The user's clock is show-

ing 14:00:20 - the new record's timestamp is twenty seconds in the past - just what the user expected.

- The user registers a new case at 23:30 local time on 01-Jul-2003. The server's local time is 00:30 on 02-Jul-2003, so it creates the case with a case start date of 02-Jul-2003. The user immediately performs a query on all cases registered on 01-Jul-2003. The newly registered case is not found.

In the second example, the server processing which records the current date as the case start date must convert from the current date (which is time zone dependent) to some fixed value that will henceforth be taken as the case start date. On the grounds of both simplicity and higher likelihood of meeting requirements, the server's local date is recorded.

The basis for how dates and date/times are handled is as follows:

- Dates are processed and displayed in a time zone-independent manner.

- Date/times are processed and displayed in the user's preferred time zone.

- The time zone of the server is used when converting from a date/time to a date (or vice versa).

The second issue was mentioned with an earlier example :- the fact that the user, on performing a search for today's cases, fails to find a record just registered. What caused this situation is as follows:

- The user carried out a transaction just before midnight, local time, on day 1. The server recorded a "start date" of day 2, based on converting it's current local date/time to a date.

- The user requested a list of transactions with a start date of day 1. Because this is a date, not a date/time the server treats it in a time zone independent manner. The newly registered record does not match the search criteria.

Searches on date/time ranges (such as the start/end of the user's local day) are only feasible if the column being searched on is itself date/time. Users will need to be aware that the current "business day" may not be the same date as the date in their local time zone. Fortunately, such situations are likely to be rare.

Notes

[1]For more information on *H2* database consult the *Cúram Third-Party Tools Installation Guide for Windows*.

Chapter 5

Data Manager

5.1 Overview

The *Data Manager* is a tool delivered as part of *IBM Cúram Social Program Management*, which allows a developer to conveniently create a database which contains a set of initial and/or test data. It is based around database independent `.xml` files so any setup done by a developer can be applied to any of the supported databases.

5.2 Intended Data Manager Process

The *Data Manager* is intended to provide assistance as part of an overall process for initial database creation. At a high level, that process includes the following three main steps:

1. Create the database, tablespaces and so on.

2. Use the *Data Manager* to create tables and complete initial data loading.

3. DBA tasks to complete database creation such as handcrafting scripts to tune the tables (ALTER) and set constraints.

The aim of the *Data Manager* is to help establish a skeletal database. Subsequently a DBA can then write handcrafted scripts to complete the database by modifying tables and settings such as LOCKSIZE or BUFFER-POOL.

> **Note**
>
> The `SQL` generated by the *Data Manager* is not intended to replace the role of a DBA. It is expected that there would be site-specific tweaking required in order to achieve production readiness.
>
> A DBA would not be expected to manipulate the Cúram model to

define extra entity options such as LOCKSIZE, BUFFERPOOL, and so on, in order for the desired SQL to be generated. This is due to a number of factors. The modeling tooling is designed to be un-aware of the final deployment environment, and DBAs would not be expected to have the skill-sets for using the modeling environment.

The *Data Manager* is not intended to be used to upgrade an existing database; it exists simply to reset the database to a known state.

5.2.1 Planning for MBCS Data

The use of multi-byte character set (MBCS) data with *Oracle*, *DB2*, or *IBM® DB2® for z/OS®* has specific database considerations, which are covered in the *Cúram Third-Party Tools Installation Guide for Windows* and *Cúram Third-Party Tools Installation Guide for UNIX*. However, for MBCS support with *DB2* or *DB2 for z/OS* specific Curam configuration is required, which impacts the behavior of the *Data Manager*.

Cúram support for MBCS data with *DB2* and *DB2 for z/OS* is enabled out-of-the-box to ensure error-free operation for users with languages requiring MBCS data and for users who find they require MBCS data when copying/pasting data from other applications. This support entails expanding the size of string columns in the database because DB2 column sizes are based on bytes, which is not necessarily the length required when MBCS data is used. This is explained in more detail in the *Cúram Third-Party Tools Installation Guide for Windows* and *Cúram Third-Party Tools Installation Guide for UNIX*. However, these default expansion settings may not be appropriate for those using only Western languages (i.e., SBCS data) and you should con-sider disabling this support or, for MBCS data, reducing the default expan-sion factor. Whether database expansion is applied by the *Data Manager* is controlled by the `curam.db.multibyte.expansion` property in `Bootstrap.properties`. The amount of expansion (a factor of 1.0 to 4.0) is set with the `curam.db.multibyte.default.factor` prop-erty in `Bootstrap.properties`. These properties are described in Ap-pendix A, *Cúram Configuration Parameters*.

To be 100% sure of no processing errors when processing MBCS data the maximum expansion factor is the default out-of-the-box. However, for many languages and data profiles it's unlikely that every database column character would require MBCS data or that all characters would require the maximum size of 4 bytes. Since there is a cost associated with using the maximum expansion factor in terms of disk space used, network overhead, memory utilization, buffer pool performance, CPU utilization, etc., it is best to use an expansion factor that balances resource utilization and perform-ance while avoiding or minimizing the possibility of application errors caused by data overruns. There are no strict rules for achieving a balance between resource utilization and the possibility of application errors; but, some considerations can help you choose a reasonable expansion factor and your testing should confirm your choice.

Depending on your language, locale, and encoding the number of required

MBCS characters will vary. For instance, if you are using English with only a few special characters (e.g. smart quotes) you will require very little expansion. Or, if you are using a language that shares the Latin alphabet with some additional characters (e.g. German) then you will need more space for MBCS data. A language (e.g. Chinese) that utilizes characters at the higher end of the Unicode range will require more space per character, which needs to be tempered by the number of characters required per word; i.e., the language may convey more information in each character than a typical Latin alphabetic character. In other words, consider the average bytes required per character, word, etc. Typically this average is only a rough estimate because, as studies have shown, character usage can vary depending on a number of factors; e.g. data context, data that is more numeric (phone numbers), versus more textual data (names) and even free-form comments. So, some additional safety factor should be considered in choosing your expansion factor.

You also have the ability to control the expansion factor at a more fine-grained level in the modeling environment by specifying the `Multi-byte_Expansion_Factor` option for a string domain and/or entity string attribute, which may be appropriate for your customizations. See the *Cúram Modeling Reference Guide* for more information on setting these options. You may need to set these fine-grained expansions at this level due to various limits within *DB2* and *DB2 for z/OS* regarding the size of rows, indexes, etc. that can be exceeded by large expansion factors (see the relevant *DB2* or *DB2 for z/OS* SQL reference for more information on these limits).

5.3 Invocation

The *Data Manager* is invoked by executing a build command of **build database**.

> ℹ️ DB2 development database optimization tip.
>
> During iterative development with *DB2* on distributed platforms the dropping and creation of tables performed during the **build database** target can be optimized to run quicker by running the script:
>
> **ant -f %CURAMSDEJ%\util\db2_optimizedbrecreation.xml**
>
> once per database. Internally this runs:
>
> **ALTER TABLESPACE USERSPACE1 DROPPED TABLE RECOVERY OFF;**
>
> **ALTER TABLESPACE CURAM_L DROPPED TABLE RECOVERY OFF;**
>
> This step should not be performed on a production database.

5.4 Database Artefacts

The *Data Manager* uses a number of generated and hand-crafted artefacts to

setup the database. This section introduces those artefacts. It does not describe the artefacts that are related rules as these are described in the *Cúram Rules Codification Guide*.

5.4.1 Data Definition XML Files

These generated files were briefly introduced in Section 3.5, *What is happening under the hood?*. The `.xml` files describe the database tables and the constraints that should be placed on them.

Example 5.1, *Table Definitions*, shows a sample table definition. An entity can have any number of `attribute` elements. Not all elements will have all the attributes (the `size` attribute is only present for strings and Large Objects).

```
<entities>
  <entity tablename="Fully qualified tablename"
    <attribute ddltype="DD Type from the UML Model"
               notnull="Indicator whether Nulls are allowed"
               size="Size qualifier for the DDL Type"
               />
  </entity>
</entities>
```

Example 5.1 Table Definitions

Example 5.2, *Foreign Key Constraints*, shows a sample foreign key constraint. There can be any number of `key`, `association` and `foreign-keypair` elements.

> **Note**
>
> If foreign keys are applied to a *DB2 for z/OS* database by the *Data Manager* manual intervention will be required to move the tables from the `check_pending` state. Please consult with your local Database Administrator (DBA) to resolve this.

```
<foreignkeys>
  <key>
    <association tablename="Local Table name"
                 othertablename="Remote table name"
                 >
      <foreignkeypair localfield="Local field name"
                      remotefield="Remote field name"/>
    </association>
  </key>
</foreignkeys>
```

Example 5.2 Foreign Key Constraints

Example 5.3, *Primary Key Constraints*, shows a sample primary key constraint. There can be any number of `key` and `attribute` elements.

```
<primarykeys>
  <key tablename="Fully qualified tablename">
    <attribute keyname="Field name"/>
  </key>
</primarykeys>
```

Example 5.3 Primary Key Constraints

Example 5.4, *Index Constraints*, shows a sample index constraint. There can be any number of `index` and `indexattribute` elements.

```
<indices>
  <index>
    <indexdetails tablename="Fully qualified tablename"
                  indexname="Name for the Index" >
      <indexattribute attribute="Field name"/>
    </indexdetails>
  </index>
</indices>
```

Example 5.4 Index Constraints

Example 5.5, *Unique Constraints*, shows a sample Unique Constraint. This can have any number of `constraint`, `association` and `attribute` elements as necessary.

```
<uniqueconstraints>
  <constraint>
    <association tablename="fully qualified tablename">
      <attribute field="field name on table for constraint">
    </association>
  </constraint>
</uniqueconstraints>
```

Example 5.5 Unique Constraints

Example 5.6, *Batch Metadata*, shows a sample of the metadata that is generated to support the batch processes that have been modeled by the developer. There may be any number of batch processes which have any number of parameters.

```
<batches>
  <batch process="Process Name"
         operation="Operation Name"
         application="Application Name"
         >
    <parameter name="Parameter name"
               type="Domain Type"/>
  </batch>
</batches>
```

Example 5.6 Batch Metadata

Example 5.7, *Security Metadata*, shows a sample of the metadata that is generated to support the security that has been modeled by the developer. There may be any number of function identifiers (FIDs).

```
<fids>
  <fid
    name="Function identifier name"
    operation="Operation to allow access to"
    fidenabled="Indicate whether enabled by default or not"
    iswebservice="Indicate whether this is a web service"
    />
</fids>
```

Example 5.7 Security Metadata

Example 5.8, *Field Level Security Metadata*, shows a sample of the metadata that is generated to support the field level security that has been modeled by the developer. There may be any number of fields returned.

```
<fieldsreturned>
    <fieldreturned
     operationname="Function identifier name"
     fieldname="Field name"
     sidname="Associated SID"
    />
</fieldsreturned>
```

Example 5.8 Field Level Security Metadata

5.4.2 Data Contents DMX Files

As well as creating the tables on the database, the *Data Manager* allows the developer to specify sample and test data which should be placed on the database. The format of the .DMX file is introduced in Example 5.9, *Data Contents File*. The developer will typically edit this file using a standard XML editor.

```
<table name = fully qualified tablename>
  <column name = column name
         type = One of:
                     number
                     text
                     bool
                     id
                     blob
                     clob
                     date
                     timestamp
                 >
  </column>
  <row>
    <attribute name = field name>
            <value>Field value</value>
    </attribute>
  </row>
</table>
```

Example 5.9 Data Contents File

The *Data Contents* DMX file is made up of a number of elements described in the following sections, some of these elements/attributes are necessary to enable customization of DMX files, described in further detail in Section 5.4.2.2, *Customizing a DMX file*.

The **table** Element

The <table> element has the following attributes:

Attribute Name	Required	Default	Description
name	Yes	None	Specifies the name of the database table.

Attribute Name	Required	Default	Description
override	No	false	Used to customize or completely override existing DMX files from within a component lower down in the SERVER_COMPONENT_ORDER.

Table 5.1 Attributes of the table Element

The `<column>` Element

The `<column>` element has the following attributes:

Attribute Name	Required	Default	Description
name	Yes	None	Specifies the name of the column.
type	Yes	None	Specifies the data type of a column. Table 5.6, *Attribute Values* describes the type that a column can be set to.
encoding	No	UTF-8	Specifies the clob data file encoding type.Check Section 5.9, *Lob Manager*.

Table 5.2 Attributes of the column Element

The `<row>` Element

The `<row>` element has the following attributes:

Attribute Name	Required	Default	Description
remove	No	false	Enables the removal of a row from a DMX file from within a component lower down in the SERVER_COMPONENT_ORDER.
locales	No	None	If omitted, the row will be applicable to all locales. If present, this must be set to a comma-separated list of locales ensuring there are no spaces between each locale. The following example indicates the `<row>` is applicable for the en and en_US locales:

Attribute Name	Re- quired	Default	Description
			`<row locales="en,en_US">.`

Table 5.3 Attributes of the row Element

The `row` element also encapsulates a collection of `attribute` elements.

The **`<attribute>`** Element

The `<attribute>` element has the following attribute:

Attribute Name	Re- quired	Default	Description
`name`	Yes	None	Specifies the name of the column.
`encoding`	No	UTF-8	Specifies the clob data file encoding type.Check Section 5.9, *Lob Manager*.

Table 5.4 Attributes of the attribute Element

> **i** Note
>
> If the number of attributes defined for a row does not match the number of columns defined the DMX processing will fail.

> **i** Note
>
> Also, when processing DMX files, the name of each attribute is not taken into account, the order is taken from the column definition at the start of the file, therefore the ordering of the attributes should match the ordering of the columns.

The `attribute` element has a required sub-element: `value`.

The **`<value>`** Element

The `<value>` element is the value to be inserted into the column for this row. For a `BLOB` the value should be a pointer to a file. To be meaningful the `name` attribute of the `attribute` element must take its value from one of the `column` elements' `name` attributes within the same DMX file. Ordering is also important as when the database is being built, database columns will be updated with content defined by the `row` elements in the order the `column` elements are listed within the DMX file.

The `<column>` elements' `type` attribute determines the valid `attribute` values. Table 5.6, *Attribute Values*, describes the relation between the column `type` and `attribute` value.

The `<value>` element has the following attributes:

Attribute Name	Required	Default	Description
language	No	None	The language attribute, along with the country attribute, make up the locale for an <attribute> element.
country	No, but if the language attribute is specified this attribute must also be specified.	None	The country attribute, along with the language attribute, make up the locale for an <attribute> element.

Table 5.5 Attributes of the value Element

> **Important**
>
> The primary key/composite key for a record must never be localized within the DMX file for that record. For example, if AddressID is the primary key for the Address table, the AddressID value element within the Address.DMX file must not be localized.

Column Type	Attribute Value
number	Value must be numeric.
text	Value must be text or multi-line text.
bool	Value must be TRUE or FALSE.
id	Value must be numeric.
blob	Value must be a relative path from the DMX file to the blob file.
clob	Value must be a relative path from the DMX file to the clob file.
date	Value must be a valid date or system date. For system date, value must be represented as SYSDATE.
timestamp	Value must be a valid time or system time. For system time, value must be represented as SYSTIME.

Table 5.6 Attribute Values

Customizing a DMX file

The Data Manager processing allows for the customization of DMX files for the initial, demo and test targets, Supported customizations include

the ability to add a row, update a row, remove a row, localize at a row/ attribute level and completely override a DMX file. This process allows for DMX files that are shipped with the Cúram application to be easily customized by adding new DMX files to new components in the relevant directory.

The DMX files to be customized must be in the following directory structure:

- `<SERVER_DIR>/components/<custom>/data/initial`

- `<SERVER_DIR>/components/<custom>/data/demo`

- `<SERVER_DIR>/components/<custom>/data/test`

To customize DMX files that are delivered out-of-the-box, new DMX files must be created and added to new components in the relevant directory within `SERVER_DIR/components/<custom>/data/initial (or /demo or /test)`.

This mechanism avoids the need to make changes directly to the out-of- the-box application, which would complicate later upgrades.

The customization process involves the merging of DMX files of the same name within the specified directory structure according to a precedence order. The order is based on the `SERVER_COMPONENT_ORDER` environment variable which contains a comma separated list of component names, the left-most having the highest priority.

> **Note**
>
> It is possible that more than one DMX file will contain data for a particular database table. As the merging of DMX files is based on file names it may be necessary to customize multiple DMX files in order to achieve a desired data customization for an individual entity.
>
> Only DMX files placed within the structure above will be included in the merging process for DMX files. If sub-directories are used within the `initial`, `demo` and `test` directories, then these will not be included in the merging process.

The merged DMX file is output to the `%SERVER_DIR%/build/datamanager/data/initial(or /demo or /test)` directory.

Rules of DMX file merging

DMX files are merged based on precedence order. There is always a more important main/source DMX file, and a file which is being merged into it. The second file is called the merge file in the following sections.

The merging rules described below are applied to decide if the rows, attributes or DMX files should be merged into the new DMX file.

- A DMX file will only be considered for merging if the new DMX file

does not have the `override` attribute on the `<table>` element set to `true`.

- A `<row>` will be inserted into the new DMX file if is determined, by using the primary key information for the record, that the `<row>` is not already present in the new file.

- If a `<row>` already exists in the new DMX file and the `remove` attribute is set to `true`, then no merging will occur. If the `remove` attribute is set to false or is not present, then the attribute values for that row will be considered for merging.

  - If the `<value>` element does not exist in the new DMX file, then the `<value>` element will be copied.

  - If the `<value>` contains a different locale, then this `<value>` entry will be copied into the new file. The locale is specified by the `language` and `country` attributes on the `<value>` element.

All examples below assume custom is before core in the `SERVER_COMPONENT_ORDER`.

The Example 1 below illustrates how merging works when using the `<table>` level `override` attribute. To use the `override` attribute, copy the contents of the existing DMX file, i.e. the core DMX file and place it in a DMX file of the same name in a `<custom>` component. Then add the following to the table element:

```
<table override="true">
```

This indicates that only DMX files in this *<custom>* component or in a component higher up in the `SERVER_COMPONENT_ORDER` will be included in the merged DMX file output produced from the Data Manager processing.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>22</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 record</value>
    </attribute>
  </row>
  <row>
    <attribute name="CONCERNID">
      <value>23</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
```

```
    <attribute name="COMMENTS">
      <value>Concern 2 record</value>
    </attribute>
  </row>
</table>
```

Example 5.10 Example 1 - Core DMX File.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN" override="true">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>55</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>My custom comment</value>
    </attribute>
  </row>
</table>
```

Example 5.11 Example 1 - Custom DMX file.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN" override="true">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>55</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>My custom comment</value>
    </attribute>
  </row>
</table>
```

Example 5.12 Example 1 - Resulting Merge DMX File.

In the resulting merge file, no rows are taken from the core DMX file as the custom DMX file is completely overriding the core DMX file through the following: <table override="true"> , resulting in all entries in the core file being excluded.

The Example 2 below illustrates how the merging process works when the <row> level remove attribute is set. To remove a row, copy the row from the existing DMX file and place it in a DMX file of the same name in a <custom> component. Then set the remove attribute on that row to true.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 core</value>
    </attribute>
  </row>
  <row>
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 2 core</value>
    </attribute>
  </row>
</table>
```

Example 5.13 Example 2: Core DMX file.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 custom</value>
    </attribute>
  </row>
  <row remove="true">
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="en">Concern 2 en custom</value>
    </attribute>
  </row>
  <row>
    <attribute name="CONCERNID">
      <value>5</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 5 custom</value>
```

```
      </attribute>
    </row>
</table>
```

Example 5.14 Example 2 : Custom DMX file.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 custom</value>
    </attribute>
  </row>
  <row remove="true">
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="en">Concern 2 en custom</value>
    </attribute>
  </row>
  <row>
    <attribute name="CONCERNID">
      <value>5</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 5 custom</value>
    </attribute>
  </row>
</table>
```

Example 5.15 Example 2 : Result merge file.

For Example 2, the `<row>` where the CONCERNID is set to 2, does not merge the `<row>` from the core DMX file. When processing the merged DMX file in Example 2, the `<row>` where the CONCERNID is set to 2 will not be included when creating the SQL insert statements, thus ensuring no entry will exist on the database for this `<row>`.

Example 3 below illustrates the setting and merging of the `language` and `country` attributes on the `<value>` element.

In this example, the COMMENTS attribute for the CONCERNID=2 has a value for the `fr` and the `en_GB` locales.

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

```
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 core</value>
    </attribute>
  </row>
  <row>
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="fr">Concern 2 French core</value>
      <value language="en"
             country="GB">Concern 2 en_GB core</value>
    </attribute>
  </row>
</table>
```

Example 5.16 Example 3: Core DMX file.

In this example, the COMMENTS attribute for the CONCERNID=2 has a value for the en locale only. The COMMENTS attribute for the CONCERNID=5 has a value for the en_US locale only.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 custom</value>
    </attribute>
  </row>
  <row remove="true">
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="en">Concern 2 en custom</value>
    </attribute>
  </row>
  <row>
    <attribute name="CONCERNID">
      <value>5</value>
    </attribute>
    <attribute name="NAME">
```

```
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="en"
             country="US">Concern 5 en_US custom</value>
    </attribute>
  </row>
</table>
```

Example 5.17 Example 3 : Custom DMX file.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 custom</value>
    </attribute>
  </row>
  <row remove="true">
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="en">Concern 2 en custom</value>
      <value language="fr">Concern 2 French core</value>
      <value language="en"
             country="GB">Concern 2 en_GB core</value>
    </attribute>
  </row>
  <row>
    <attribute name="CONCERNID">
      <value>5</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="en"
             country="US">Concern 5 en_US custom</value>
    </attribute>
  </row>
</table>
```

Example 5.18 Example 3 : Result merge file.

In Example 3 above, for the <row> where the CONCERNID is set to 2, the resulting merge file has values for the en, fr and the en_GB locales, i.e. a merge of both core and custom <value> elements.

Example 4 below illustrates the <row> level locales attribute.

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

```
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 core</value>
    </attribute>
  </row>
  <row locales="en_GB">
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="fr">Concern 2 French core</value>
      <value language="en"
              country="GB">Concern 2 en_GB core</value>
    </attribute>
  </row>
</table>
```

Example 5.19 Example 4: Core DMX file.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 custom</value>
    </attribute>
  </row>
  <row locales="en,en_US">
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="en">Concern 2 en custom</value>
    </attribute>
  </row>
</table>
```

Example 5.20 Example 4 : Custom DMX file.

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

```
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 custom</value>
    </attribute>
  </row>
  <row locales="en,en_US">
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="en">Concern 2 en custom</value>
      <value language="fr">Concern 2 French core</value>
      <value language="en"
             country="GB">Concern 2 en_GB core</value>
    </attribute>
  </row>
</table>
```

Example 5.21 Example 4 : Result merge file.

In Example 4 above, the value for the `locales` attribute is taken from the row in the component that is higher up in the SERV-ER_COMPONENT_ORDER, i.e. the custom component.

The primary key/composite key for a record is used to determine the over-riding/merging process for DMX files. DMX files will be merged based on the definition of the primary key for the table/entity the DMX file represents. For all modelled entities, the primary key information is stored in the generated `<SERVER_MODEL_NAME>_PrimaryKeys.xml` file in the build directory, i.e. `%SERVER_DIR%/build/svr/gen/ddl`. For all non-modelled components, the primary key information for entities must be stored in a file called `<SomeName>_PrimaryKeys.xml` within `%SERVER_DIR%/components/<custom>/data/ddl` directory. If this file is named correctly in the specified location, the DMX processing will contain the relevant primary key information for the non-modelled component.

Retrieving values from DMX files for database insertion

The Data Manager uses the `<row>` level `remove` attribute to determine if an entry will be inserted onto the database for that row. If the `remove` attribute is set to `true`, then the Data Manager will not insert an entry for that row. The row will be ignored.

DMX files store the locale information for the attributes for the database table. As the database must be built for only one locale, the Data Manager

uses the `curam.dmx.locale` property to determine the locale that must be used when inserting data specified in DMX files onto the database. This property can be set in either the `Bootstrap.properties` file or as a system variable. If set in both the `Bootstrap.properties` file and as a system variable, the system variable will override the setting in the `Bootstrap.properties` file. This property must be set to a valid locale, i.e. in the format language_Country, where language is mandatory and country is optional. For example,

```
curam.dmx.locale=en_US
```

If this property is not set, the infrastructure will fallback on the `en` locale.

As mentioned, the Data Manager processing uses the `curam.dmx.locale` to determine the value to insert for an attribute in a DMX file. The locale can be specified at a `<row>` or `<attribute>` level. If specified at a row level, then this takes precedence over the attribute level. For example, given the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CONCERN">
  <column name="CONCERNID" type="id"/>
  <column name="NAME" type="text"/>
  <column name="COMMENTS" type="text"/>
  <row>
    <attribute name="CONCERNID">
      <value>1</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value>Concern 1 core</value>
    </attribute>
  </row>
  <row locales="en_GB">
    <attribute name="CONCERNID">
      <value>2</value>
    </attribute>
    <attribute name="NAME">
      <value/>
    </attribute>
    <attribute name="COMMENTS">
      <value language="fr">Concern 2 French core</value>
      <value language="en"
              country="GB">Concern 2 en_GB core</value>
    </attribute>
  </row>
</table>
```

In this example, if the `curam.dmx.locale` environment variable is set to the `fr` locale, then there will be no entry inserted for the record where CONCERNID is set to 2, as the `locales` attribute for the `<row>` is only applicable for the `en_GB` locale, even though the attribute for COMMENTS has an entry for the `fr` locale.

The Data Manager attempts to match the locale specified by the `curam.dmx.locale` environment variable with the `locales` attribute for the `<row>` element within a DMX file. If this attribute is not set, then the Data Manager attempts to match on the `<value>` for an

<attribute>, i.e. it tries to match on the `language` and `country` attributes of the <value> element.

Since DMX files are not guaranteed to contain an entry for every locale, a fall back mechanism is in place. This fallback mechanism is only applicable to the attribute <value> element, i.e. it is not applicable to the <row> `locales` attribute. Once a <value> has been found and there is no direct match with the locale specified by `curam.dmx.locale`, the rules for fall back are as follows:

* If the `curam.dmx.locale` is set to include a language and country part, the processing looks for an attribute where the `language` and `country` attributes are set on the <value> element. If this is not found, then the `country` is removed and the search looks for a <value> where the `language` attribute matches, if this is not found, then the search looks for a <value> that does not have the `language` and `country` attributes set, i.e. a default match. If this is not found, then no entry is inserted onto the database for this <value>.

```
<row>
    <attribute name="ADDRESSELEMENTID">
      <value>3227</value>
    </attribute>
    <attribute name="ELEMENTTYPE">
      <value language="en">EN_TYPE</value>
      <value country="US" language="en">EN_US_TYPE</value>
    </attribute>
    <attribute name="ELEMENTVALUE">
      <value language="fr">French Value</value>
    </attribute>
</row>
```

Example 5.22 Locale Fallback Example

In Example 5.22, *Locale Fallback Example*, lets assume the `curam.dmx.locale` is set to en. The following is set for each attribute:

* ELEMENTTYPE - EN_TYPE is the value inserted onto the database for this attribute, as this is the value set for the en locale.

* ELEMENTVALUE - null is inserted onto the database for this attribute. This attribute has the `language` attribute set to `fr`. The locale that is being searched for is en, a value for en is not found, so a <value> that contains no `language` or `country` attributes is searched for, i.e. the default value, as this does not exist, null is inserted for this attribute.

Validation of DMX files

All DMX files in `%SERVER_DIR%/components/componentName/data` directories will be validated against a DMX schema file when the build database target is run. This schema file is located in `%CURAMSDEJ%/lib/DMX.xsd`. For any DMX file that is not in the correct format, a warning will be displayed.

The validation of DMX files is controlled by the `curam.dmx.disable.validation` system variable. Validation is enabled by default, to disable the validation, this system variable should be passed into the database build setting it to `true`, as follows:

```
build database -Dcuram.dmx.disable.validation="true"
```

The ability to treat these warnings as errors is available by setting the `prp.warningstoerrors` property. If this is set to `true`, the warnings will be treated as errors and the build database will fail.

Tracing Information for the DMX Merging Process

It is possible to turn tracing on for the DMX merging process. This can help assist in debugging any issues that may occur, as a result of merging DMX data. The system property `curam.dmx.tracing`, if set to `true`, produces tracing information to the console for the DMX file being processed. This property is `false` by default.

The tracing output includes:

- The name of the file being processed;

- The key value for a row that is being merged (only where duplicate rows exist);

- Information indicating the merging process has finished for a DMX file.

The following is an example of setting this property:

```
build database -Dcuram.dmx.tracing=true
```

Example 5.23 Set tracing for DMX files.

When set to `true`, this property outputs a large amount of data to the console and must therefore only be used for debugging purposes.

5.5 Database Object Naming

Typically the names of the objects on the database are clearly visible from the *Data Manager* XML Files (for example, table names and column names). The *Data Manager* does provide support for the naming of objects which are not directly visible in these files.

5.5.1 Short Name Substitution

The Short Name Substitution feature will be removed in a future version of *IBM Cúram Social Program Management*. The third party databases now supported no longer have the SQL identifier limitations that originally necessitated the feature. Consequently, it is no longer necessary to use this fea-

ture and it has been removed from the product documentation. If you still require this feature please contact Support for the information that previously was available in this document. Please refer to the *Cúram Supported Prerequisites* document for comprehensive details of the supported versions of database management systems.

5.5.2 Primary Key Indices

By default the primary key index will have the same name as its corresponding table.

If required, a prefix can be specified for the primary key index name using the generator command line option **-primarykeyindexprefix**. For example setting the property `ex-tra.generator.options=-primarykeyindexprefix PI_` in `Bootstrap.properties` will result in the primary key index for a table named Person being named PI_Person. If the index name length is greater than the SQL identifier limit supported by your database you will encounter an error during SQL processing.

5.5.3 Primary Key Constraints

By default, the generated DDL for adding a primary key to a table takes the form:

alter table TTTT add primary key (AAAA)

where

TTTT is the table name.

AAAA is a comma-delimited list of the primary key attributes.

By specifying the command line option **-usenamedprimarykeyconstraint** through the **extra.generator.options** this DDL can be made take the form:

```
alter table TTT add constraint CCCC primary key
      (AAAA)
```

where

CCCC is the name of the primary key constraint.

In this case the name of the primary key constraint defaults to the same as the name of its corresponding table. Also, like primary key index names, a prefix can be applied to this name using the **-primarykeyconstraintprefix** command line option. If the constraint name length is greater than the SQL identifier limit supported by your database you will encounter an error during SQL processing.

5.5.4 Tablespaces

i Note

This section is specific to *DB2 for z/OS*.

By default the behavior is for tablespaces to be created implicitly during table creation. The exceptions to this are:

1. The tablespace named by the `curam.db.zos.32ktablespace` property is created explicitly by the datamanager and tables exceeding the 4K row limit are placed in this tablespace.

2. Tablespaces identified in the `Tablespace.properties` file are created explicitly by the datamanager. If the table specified for the tablespace exceeds the 4K row limit the tablespace is defined in the 32K BUFFERPOOL. Otherwise, it will take the default setting.

When using the `Tablespace.properties` file the format of the entries is:

```
tablename=tablespacename
```

Comments are specified by the "#" character in column one.

> **Note**
>
> If the tablespace for the table that exceeds the 4K row limit is defined in the `Tablespace.properties` file then this tablespace will be used over the one defined in the property `curam.db.zos.32ktablespace`.

> **Note**
>
> When using *DB2 for z/OS* version 8 the use of the default 32K tablespace (`curam.db.zos.32ktablespace`) can result in SQLCODE -913 errors during login, but could also occur in other contexts. To avoid these errors you should do one of the following:
>
> 1. Ensure your Cúram default 32K tablespace is segmented (*SEGSIZE*; see the *DB2 Universal Database for z/OS SQL Reference Version 8* for more information).
>
> 2. Explicitly define tablespaces for each Cúram table that defaults to the 32K tablespace (e.g. `SELECT * FROM SYSIBM.SYSTABLESPACE WHERE NAME = <curam.db.zos.32ktablespace value>`) and assign each table to a specific tablespace via the `Tablespaces.properties` file.
>
>   (This is not an issue when using *DB2 for z/OS* version 9 because tablespaces are segmented by default.)

> **Note**
>
> In *DB2 for z/OS* version 9 the behavior of the `ALTER TABLE DROP PRIMARY KEY` SQL statement changed as follows: "If the table space was implicitly created, the corresponding enforcing in-

dex is dropped if the primary key is dropped." Most production users would typically explicitly create their tablespaces and would not be impacted by this change, but in test environments this may not be the case. The symptom of this issue is an SQLCODE -551 error on a `DROP INDEX` statement following the `ALTER TABLE DROP PRIMARY KEY` statement. To avoid this error you can either:

- Manually remove the generated `DROP INDEX SQL` statement from the *Data Manager*-generated SQL to take into account the new behavior; or,

- Explicitly define the tablespace and specify it in the `Tablespace.properties` file. For example, for the `USERS` table, your `Tablespace.properties` file would contain:

```
USERS=USERSTS
```

5.6 Data Manager Configuration

Typically the *Data Manager* sets up the database from a number of different components:

- SDEJ Tables

- Application Tables

- Initial Data

- Demo Data

- Test Data

The selection of which set of data to apply effectively depends on the task the developer wishes to perform.

The *Data Manager* is configured using the `datamanager_config.xml` configuration file. The file is located at:

```
SERVER_DIR\project\config\datamanager_config.xml
```

The structure of `datamanager_config.xml` is shown in Example 5.24, *Data Manager Configuration*.

```
<datamanager>
  <compositetarget name="target name">
    <subtarget name="subtarget name"/>
  </compositetarget>
  <target name="subtarget name">
    <entry name="relative filename or relative directory"
           type="sql, DMX or xml"
           base="sdejscripts or basedir"/>
  </target>
```

```
</datamanager>
```

Example 5.24 Data Manager Configuration

The file is organized as follows:

Target Tag

> This has a name attribute specifying the name of the target and a set of associated entry tags.

Entry Tag

> This has three attributes associated with it.

> **Name Attribute**

>> This specifies the file or directory associated with this attribute and its offset from the base attribute.

> **Type Attribute**

>> This specifies whether the file is an SQL script, a `.DMX` file or an `.xml` file.

> **Base Attribute**

>> This specifies the system dependent offset of the file on the local machine. It may be specified as one of `basedir` (the directory above the Data Manager) or `sdescripts` (the location of the SDEJ installation).

Any of the targets listed in this configuration file can be passed to the **build database** target.

The `datamanager_config.xml` file is used when running the the **build database** target. When this target is run, composite targets specified within the `datamanager_config.xml` can be called. By default, the **all** composite target is called within the `datamanager_config.xml` file. To call a different composite target the `prm.target` can be passed to the **build database** target specifying the composite target to be called. For example, to call the initial composite target, the following could be executed:

```
build database -Dprm.target=initial
```

New composite targets can be added to the `datamanager_config.xml` file. The composite target can contain any number of subtargets. The following is an example of specifying a new composite target *mycompositetarget* that calls *mynewtarget*.

```
<target name="mynewtarget">
  <entry base="basedir"
      name="components/core/data/initial/
            handcraftedscripts/NewScript.sql"
      type="sql"
  />
</target>
<compositetarget name="mycompositetarget">
  <subtarget name="mynewtarget"/>
```

```
</compositetarget>
```

5.7 Database Synchronization

Typically the Data Contents XML files are hand crafted by a developer. However the infrastructure provides *Ant* targets to create a Data Contents XML file from the database. The *Data Extractor* is invoked by executing a build command of **build extractdata**. By default the full database is extracted and DMX files are created for any tables that contain data. An optional parameter of `tablename` can be passed to specify that only one or more tables should be extracted e.g. **build extractdata -Dtablename=Users**. If you want to extract multiple tables during the one run, pass a comma separated list of tables to the `tablename` parameter.

The generated `.DMX` files are placed in a `%SERVER_DIR%/build/dataextractor` folder. Under this folder the contents of any clobs or blobs are also extracted and stored in a file which is based on the naming format: `<tablename><rownumber>`.

5.8 Statistics

Databases use an optimizer to determine the most efficient access path to data on the database. The optimizer uses statistics about the physical characteristics of a table and the associated indexes to determine this information. These characteristics include number of records, number of pages, and average record length. If no statistics are available on the database, then the optimizer makes a guess as to the best access path to use and this can often lead to performance issues, including unnecessary deadlock and timeout exceptions. The **runstatistics** target is available to gather these necessary statistics on the database and will be run against all Cúram database tables.

> **Note**
>
> The "runstatistics" target is not supported with *DB2 for z/OS* due to the architectural differences of this platform. Consult with your local database administrator in regard to invoking the equivalent *DB2 for z/OS* functionality.

5.9 Lob Manager

The LOB Manager is part of the Data Manager which enables Clobs and Blobs to be loaded onto the database.

In the data contents file Blob and Clob fields are handled slightly differently to other fields, in that the `value` element will not contain the literal data but will instead contain a reference to a file containing the data.

The Example 5.25, *Blob Data Contents File*, illustrates how a table with a numeric and blob column can be populated with one record using a binary

file from disk.

```
      <table name = "BlobEntity">
         <column name = "imageID" type = "number"/>
         <column name = "imageData" type = "blob"/>
         <row>
      <attribute name = "imageID">
      <value>1</value>
            </attribute>
         </row>
         <row>
       <attribute name = "binaryData">
       <value>./images/1.jpg</value>
       </attribute>
   </row>
      </table>
```

Example 5.25 Blob Data Contents File

Note that to load Blobs, the LOB Manager can only be used on tables for which the primary key fields are known. This is because inserting a LOB involves an SQL insert followed by an SQL update, and the SQL update must be capable of addressing a single record by means of its primary key.

The Example 5.26, *Clob Data Contents File*, illustrates how a table with a numeric and clob column can be populated with one record using a character data file from disk.Here, the clob data file is encoded with UTF-16 format, and this is specified in the attribute element with encoding as UTF-16 for that row, so the clob content gets encoded before it gets inserted.

```
      <table name = "Entity">
         <column name = "ID" type = "number"/>
         <column name = "content" type = "clob"/>
         <row>
      <attribute name = "ID">
      <value>1</value>
            </attribute>
         </row>
         <row>
       <attribute name = "content" encoding = "UTF-16">
       <value>./clobcontentdir/1.txt</value>
       </attribute>
   </row>
      </table>
```

Example 5.26 Clob Data Contents File

The Example 5.27, *Clob Data Contents File in encoded format*, illustrates how a table with a numeric and clob column can be populated with two records using the character data files from disk.Here, if all the clob data files are encoded in UTF-16 format, then this can be specified at column level, using encoding attribute, so all the rows for clob type uses the same encoding type of that column. To override this for only a single row, the encoding type can be specified as in previous example at attribute element level of that row element.

```
        <table name = "Entity">
          <column name = "ID" type = "number"/>
          <column name = "Data" type = "clob"
                encoding = "UTF-16"/>
          <row>
      <attribute name = "ID">
      <value>1</value>
            </attribute>
        </row>
        <row>
       <attribute name = "Data">
       <value>./clobcontentdir/4.txt</value>
       </attribute>
  </row>
        <row>
      <attribute name = "ID">
      <value>2</value>
            </attribute>
        </row>
        <row>
       <attribute name = "Data">
       <value>./clobcontentdir/2.txt</value>
       </attribute>
  </row>
        </table>
```

Example 5.27 Clob Data Contents File in encoded format

The LOB manager identifies primary keys by means of the `dataman-ager_config.xml` file, so this file must contain a reference to the generated `_PrimaryKeys.xml` relating to table containing the LOB.

Chapter 6

SQL Checker

6.1 Overview

The *IBM Cúram Social Program Management* SDEJ produces a database access layer which is based around *JDBC*. JDBC is dynamic SQL from the viewpoint of database and as such there is no ability to check the syntax and semantics of the statements prior to their first execution. The SQL checker provides a method of validating the syntax and semantics of these SQL statements before they are first exercised.

6.2 Under the Hood

The SQL checker is invoked by an *Ant* target and generates a simple *Java* program which uses SQLJ rather than JDBC. This program is generated into `/build/sqlcheck/SQLJTemp.sql`. This *Java* program contains all the elements that should be checked, namely the hand crafted SQL in the model and the Data Manager. Because SQLJ is static SQL the program can be compiled in advance of deployment, provided the database is already created and populated.

The SQL checker can also check the contents of the model for database portability. This is useful in situations where primary development is against one kind of database (for example *DB2*) but final deployment is against another database (for example *DB2 for z/OS*). The elements checked for include:

- Comparison of Host Variables to NULL

This check is performed because hand-crafted SQL can use the SQL `is Null` keyword on a host variable. If this is done the Cúram Generator automatically produces a cast to the correct fundamental SQL datatype for the database that is being built against. However, this means that the resultant `.ear` file cannot be deployed against a database of a different type unless it

is completely re-built.

6.3 Limitations

The SQL Checker is designed to reduce the number of syntax and portability errors that remain until deployment as this reduces the effort expended in testing for and removing these errors. However, it is not a replacement for a comprehensive test suite as it does not catch all the possible errors. There are a number of reasons for this:

Reliance on the SQLJ Check

> The SQL Checker is only as good as the SQLJ compiler that it invokes. Any syntactical or semantic errors which are not reported by the compiler will not be reported by the SQL Checker.

Portability Warnings

> The SQL Checker is only designed to capture and report the most common portability errors. It is not a replacement for early and comprehensive testing on the final target database.

Limitation with H2

> H2 doesn't provide an implementation of an SQLJ checker; therefore, it only performs a portion of the perceived checks that the SQL Checker does.

Chapter 7

Eclipse

7.1 Overview

Eclipse is the core IDE for development of *IBM Cúram Social Program Management*. It is the underlying technology in:

- *IBM® Rational® Application Developer for WebSphere*;

- *Rational Software Architect*;

- *Rational Software Architect for WebSphere*.

This chapter describes relevant aspects of *Eclipse* as well as providing some tips and tricks. It does not attempt to describe the general features or usage of *Eclipse*; e.g., the *Java* Editor or debugging as that information is provided by the vendor, see http://www.eclipse.org/ for more information.

The term "*Eclipse*" which is used throughout this chapter applies to all supported tooling based on *Eclipse*; e.g., *Rational Software Architect*.

The supported version of *Eclipse* or its usage through the *Rational* product versions can be found in the *Cúram Supported Prerequisites*.

7.2 Curam Projects in Eclipse

Four projects are provided that should be imported into *Eclipse*:

Project Name	File System directory	Contents
CuramSDEJ	CuramSDEJ	The Server Development libraries.
CuramCDEJ	CuramCDEJ	The Client Development libraries, depends on the CuramSDEJ project.

Project Name	File System directory	Contents
EJBServer	EJBServer	The Cúram Server application, depends on the CuramSDEJ project.
Curam	webclient	The Cúram Client application, depends on the CuramCDEJ project.

Table 7.1 Transaction settings

Dependencies allow for exposed jar libraries in referenced projects to be used in code developed in the dependent project.

The `CuramCDEJ` and `CuramSDEJ` are non-development projects that are only containers for libraries. All development should be done within the `EJBServer` and `Curam` projects.

7.3 Eclipse Configuration Files

Each *Eclipse* project is configured through two XML files; a `.project` and a `.classpath` file. Also a number of preferences and settings can be configured at a project level rather than workspace level; the effect of setting these at a project level means that this configuration, which form files and entries in a .settings folder under the project, can be distributed which the project in a team environment.

The configuration mentioned in section is maintained by right-clicking on a Project within the Project Explorer view in *Eclipse* and selecting `Proper-ties`.

7.3.1 .project File

The `.project` file holds the project nature and builders and for a typical *Java* project holds a single nature and builder corresponding a *Java* project. Additionally in the `Curam` project there is a *Apache Tomcat* nature to signify the project can be configured for and deployed on *Tomcat*. The project's dependencies are also maintained in the `.project` file.

7.3.2 .classpath File

The `.classpath` maintains the Project's source and target references for *Java* compilation and jar or project dependencies.

This configuration is maintained through the *Java Build Path* page in the Project's properties. Source entries can be added, ordered or new jar file dependencies can all be managed through this page.

Optionally, Access Rules and JavaDoc references can be configured on jar

files. Access Rules are discussed further in Section 7.4, *Access Rules*.

7.3.3 .settings Directory

The `.settings` folder maintains a number of the other preferences that can be maintained at the project level e.g. Compiler warning/error levels, Code style settings, etc. The preference pages offering this ability to maintain at a project level can be seen to have an `Enable project specific settings` at the top of the page.

This directory can be added to SCM control and settings distributed to team members as required.

7.4 Access Rules

The `Access Rules` option allows jar files within an *Eclipse* project `.classpath` to define an access level for packages and classes. There are three different levels of access: non-accessible, discouraged and accessible. When the compiler within *Eclipse* detects access to a type that should not be accessed, it will create a problem marker rather than compile failure:

- Non-accessible rules define types that must not be referenced. The compiler typically creates an error marker for accesses to these types;

- Discouraged rules define types that should not be referenced. The compiler typically creates a warning marker for accesses to these types;

- Accessible rules define types that can be referenced.

Access rules are applied and provided rules for a number of the jar files in the `.classpath` files of the *Eclipse* projects. These access rules complement each jar file's API and through the `accessible rule` indicate access that is compliant[1] as per the *Cúram Development Compliancy Guide*. Classes defined as `non-accessible` or `discouraged` are not supported for usage and are subject to change without notice and may not respect their API; hence they can impact the ability to easily integrate *IBM Cúram Social Program Management* upgrades.

> **Note**
>
> A large number of discouraged accesses exist in the out of the box Social Program Management Platform that may have been copied into your codebase as part of sub-classing or extension work. In a future release it is expected that these accesses will be removed and appropriate alternative APIs provided where none currently exist. In order to reduce future impact to your codebase, in regard access to discouraged code, you should treat these accesses as non-accessible and work to seek an alternate API.

7.5 Working Sets

A common problem in *Eclipse* is that as the content in your workspace grows it can be overwhelming to navigate through all the directories and difficult to focus on the areas of interest to you. *Eclipse* solves this through Working Sets which is a way to specify, in a global location, which working set you are currently interacting with. The following views and dialogs in *Eclipse* support the concept of working sets:

• The Navigator;

• The Package Explorer;

• The Projects View;

• The Packages View;

• The Types View;

• The Problems View;

• The Open Type Dialog.

Working sets can be especially useful for example on the Problems View, in terms of viewing what problems relate to your owned code. The following steps detail how to set a working set on the Problems view to only display problems related to the `custom` component:

1. From the *Problem View* menu select *Configure Contents*;

2. In the *Configure Contents* dialog you must first add a filter from the *Configurations* panel. Click the *New...* button and name this filter (e.g. `Custom`) and click *OK*. This will create the filter checking it in the *Configurations:* list. Under *Scope:* select the *On Working Set: Window Working Set* radio button and click the *Select...* button to add a new working set;

3. In the *Select Working Set* dialog select the *Selected Working Sets* radio button and click the *New...* button;

4. The *New Working Set* wizard can then be used to add types to the working sets. In this instance we want to add a *Java* type and select the custom source directory.

5. In the *Select a working set type* panel, select *Java* from the *Working set type:* and click the *Next >* button. In the *Java Working Set* panel, select items in the *Workspace content:* list and add them to the *Working set content:* list using the *Add -->* button. Use the other buttons in the list to manage the *Workspace content:* list. Specify a name in the *Working set name:* text box. Click the *Finish* button. You can invoke the *New Working Set* wizard again to create more working sets. Before clicking the *OK* button to exit the Wizard ensure your *Selected Working Sets* are checked.

6. On clicking *OK* to exit the *Configure Contents* dialog your Problems View will be updated to only display errors, warnings or informationals

relating to the newly created `Custom` filter.

Notes

[1]Access Rules can only be applied to jar files so should not be treated as a complete solution to police compliancy.

Chapter 8

Logging

8.1 Overview

Logging facilities in an *IBM Cúram Social Program Management* application are provided by the `curam.util.resources.Trace` class which provides a convenient wrapper onto the *Apache log4j* [1] API.

This allows developers to log any information without concerning themselves with whether the program is being run in on-line or batch mode. The final destination of the trace information is highly configurable and may be a log file associated with the application server, a standalone log file, a console or even a database.

8.2 Usage

The main interface into the tracing API is through an instance of the `org.apache.log4j.Logger` class. The infrastructure provides a number of named instances which match the categories described in Table 8.1, *Logging Hierarchy*. The top level category is accessed through `curam.util.resources.Trace.kTopLevelLogger` as shown in Example 8.1, *Usage of the loggers*.

```
curam.util.type.DateTime timeNow;
timeNow = curam.util.type.DateTime.getCurrentDateTime();
curam.util.resources.Trace.kTopLevelLogger.info(
        "This function was called at ");
curam.util.resources.Trace.kTopLevelLogger.info(timeNow);
```

Example 8.1 Usage of the loggers

It should be noted that the above code produces two trace records. This will not be easily visible if *log4j* is configured to use a flat file or the console. However if a *log4j* viewer is used then the two trace records will result in a needless entry which will complicate the view without any added benefit.

As such it is recommended that trace statements which contain logically dependent data are performed in a single call.

A formatted textual representation of a Cúram struct class object may be obtained through a call to the class `curam.util.resources.Trace.objectAsTraceString` call. For example:

```
curam.util.struct.ProcessNameKey someKey =
  new curam.util.struct.ProcessNameKey;
someKey.processName="someValue";

curam.util.resources.Trace.kTopLevelLogger.info("DEBUG\n");
curam.util.resources.Trace.kTopLevelLogger.info(
  curam.util.resources.Trace.objectAsTraceString(someKey));
```

Example 8.2 Tracing a Cúram Struct

8.3 Logging Hierarchy

The Cúram infrastructure produces trace records in specific categories with specific levels. This allows them to be easily filtered in a *log4j* viewer. The categories and levels supported are described in the following table where *<BPO>*, *<Entity>* and *<Facade>* are the names of the relevant Cúram class. The *<CodePackage>* field is left empty if the class is not located in a code package.

Category	Level	Meaning
Trace	Error	Loggable exceptions which have not been caught in the code.
Trace.BatchLauncher	Info	Progress of Batch Launcher
Trace.BatchLauncher	Error	Errors in Batch Launcher
Trace.CodeTable	Debug	Tracing information about code table lookups.
Trace.DataAccess.<Entity>	Info	SQL statements executed by entity objects.
Trace.DataAccess.<Entity>	Debug	Results of SQL select statements.
Trace.Methods.<CodePackage>.<BPO>	Info	Business Object method invocation.
Trace.Methods.<CodePackage>.<BPO>	Debug	Arguments and types of arguments for Business Object method invocation.
Trace.Rules	Info	Progress of Rules Engine.
Trace.ServerCalls.<CodePackage>.<Facade>	Info	Server method invocations by remote clients.

Category	Level	Meaning
Trace.ServerCalls.<CodePackage>.<Facade>	Debug	Arguments and types of arguments for server method invocation.
Trace.Tools	Info	Progress of build time tools. E.g: configtest
Trace.Tools	Warning	Warnings from build time tools
Trace.Tools	Error	Errors from build time tools

Table 8.1 Logging Hierarchy

8.4 Logging Level

When logging the Cúram server, trace level should be taken into consideration. These settings can be used to guard the calls made into *log4j* to improve the performance in environments where tracing is not required[2].

The current level of tracing can be checked by calling the method:

```
curam.util.resources.Trace.atLeast(Trace t)
```

where the parameter to this method can be one of the following:

- `curam.util.resources.Trace.kTraceOff`

- `curam.util.resources.Trace.kTraceOn`

- `curam.util.resources.Trace.kTraceVerbose`

- `curam.util.resources.Trace.kTraceUltraVerbose`

The trace level for your application can be specified by setting the `curam.trace` property as defined in Chapter 4, *Cúram Configuration Settings*. Valid values for this property are:

- `trace_on`

- `trace_verbose`

- `trace_ultra_verbose`

The amount of logging done by your application code should reflect the current logging level of the application. The following code extract demonstrates this:

```
if (curam.util.resources.Trace.atLeast(
                    curam.util.resources.Trace.kTraceOn)) {
    curam.util.resources.Trace.kTopLevelLogger.info(
      "hello world.");
}
```

Example 8.3 Logging example in application code

The Cúram infrastructure provides support for a number of standard trace options which provide a convenient view on top of the trace levels. All of the options result in significant information being written to the log and will have a significant impact on the performance of the application. The following are the properties that may be set as described in Chapter 4, *Cúram Configuration Settings*, and the level at which they are set at default (O is On, V is Verbose, U is Ultra).

Property Name	Meaning	Enabled
curam.trace.servercalls	Trace server method invocations by remote clients. This includes the name of the user requesting the invocation.	O
curam.trace.methods	Trace all business object method invocation.	V
curam.trace.method_args	Dump arguments, including their types, to BO method invocations.	U
curam.trace.sql	Trace SQL statements executed by entity objects.	V
curam.trace.sql_args	Dump results of SQL select statements.	U
curam.trace.rules	For more information refer to the Runtime Rules Logging in the *Cúram Rules Codification Guide*.	U
curam.trace.smtp	Trace the messages that are sent to the mail server.	

Table 8.2 Diagnostic Tracing Options

8.5 Configuration

log4j provides extensive support for configuring the destination of the trace information. This section does not attempt to duplicate the *log4j* documentation but places this information in the context of *IBM Cúram Social Program Management*. The configuration information should be placed in a file pointed at by the `curam.trace.configfile.location` property.

If the `curam.trace.configfile.location` property is not set, the default *log4j* setting is to use a Console Appender. The Console Appender simply outputs everything output at the default (or higher) *log4j* level to System Out. The default *log4j* level for the top level logger (and all inherited loggers) is set to DEBUG. [3]

Example 8.4, *Configuring log4j* will result in trace information being writ-

ten to a rolling file appender. This means the output is placed in a file until it reaches a specified size. Once it reaches this size it is "rolled-over", and it is renamed by appending a `.1` to the file name. If a `.1` file exists it is first renamed to `.2` and so on.

This is suitable for development environments where a historical trace can be useful.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<!--
  | For more configuration information and examples
  | see the Jakarta Log4j website:
  | http://jakarta.apache.org/log4j
-->

<log4j:configuration
  xmlns:log4j="http://jakarta.apache.org/log4j/"
  debug="false">

  <!-- ========================= -->
  <!-- Append messages to a File -->
  <!-- ========================= -->
  <appender name="OutputToFile"
            class="org.apache.log4j.RollingFileAppender">
    <param name="File"
           value="d:/CuramProps/CuramAppLog.log" />
    <param name="Threshold"
           value="debug"/>
    <param name="MaxBackupIndex"
           value="3"/>

    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
           value="[%-5p] [%d{dd MMM yyyy HH:mm:ss}] [%c] - %m%n"/>
    </layout>
  </appender>

  <!-- ====================== -->
  <!-- Setup the Root category -->
  <!-- ====================== -->
  <root>
    <level value="INFO"/>
    <appender-ref ref="OutputToFile"/>
  </root>

</log4j:configuration>
```

Example 8.4 Configuring log4j

There are a number of customizable values in this file:

- The name of the log file is set to be `d:/CuramProps/CuramAppLog.log`.

- The maximum number of previously rolled back files which are preserved is set to 3.

- The maximum file size is not explicitly set so the default of 10Mb is used.

- The Conversion pattern means the following is output:

  - %-5p : The level of the trace message after being left padded to be a 5 character string.

- • %c : The category of the trace message.

- • %m : The trace message itself.

- • %n : A platform specific line separator.

- • The *log4j* level is set to INFO, which means that all items logged at the DEBUG level will be ignored. This overwrites the default level of DE-BUG set by the infrastructure.

However, direct access to a file may not be an ideal mechanism if the trace output should be monitored. Example 8.5, *Configuring log4j to log to a socket* will result in trace information being written to a socket. A listener (such as *Apache Chainsaw* which is delivered with *log4j*) can then be used to display the resultant information.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<!--
   | For more configuration information and examples
   | see the Jakarta Log4j website:
   | http://jakarta.apache.org/log4j
-->

<log4j:configuration
  xmlns:log4j="http://jakarta.apache.org/log4j/"
  debug="false">

  <!-- =========================== -->
  <!-- Append messages to a Socket -->
  <!-- =========================== -->
  <appender name="OutputToSocket
            class="org.apache.log4j.net.SocketAppender">
    <param name="RemoteHost"
           value="localhost" />
    <param name="Port"
           value="4445"/>

    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
          value="[%-5p] [%d{dd MMM yyyy HH:mm:ss}] [%c] - %m%n"/>
    </layout>
  </appender>

  <!-- ====================== -->
  <!-- Setup the Root category -->
  <!-- ====================== -->
  <root>
    <level value="INFO"/>
    <appender-ref ref="OutputToFile"/>
  </root>

</log4j:configuration>
```

Example 8.5 Configuring log4j to log to a socket

The Conversion pattern used in this file is the same but some extra customizable values have been introduced:

- • The host name and port of the remote server are set to localhost and 4445 respectively.

Numerous other possibilities exist for this configuration and this section

does not attempt to duplicate the existing *log4j* documentation. However, it is worth noting that *Nested Diagnostic Contexts* are not currently supported.

8.6 Statistics

Tracing facilities are provided to allow server-related information and diagnostics to be output to a central location. It is possible to use this information to collect performance information about client visible Cúram server functions, i.e. any operations invoked by the Cúram web client. However writing trace informational typically has an impact on performance, because the *log4j* appender should always be configured to maintain the contents after a server crash (for example buffered file access should not be used). For performance benchmarking it is highly desirable that the benchmarking process should not itself impose a performance overhead on the application being measured. For this reason, A way to collect server function performance statistics is provided that imposes less overhead than server tracing, and which produces output in a format more amenable for automated processing as part of benchmark analysis.

To avoid performance overhead on the server output is written to separate log files, one per Session Bean (Cúram Facade) in the application. Each log file has an associated 4Kb memory buffer, so there is a memory overhead imposed by the collection of server benchmarks. It is assumed that a realistic benchmark configuration will involve application server machines with a significant amount of physical memory.

The statistics files are created in the directory specified by the `curam.test.trace.statistics.location` property if the `curam.test.trace.statistics` property is set. They are named `<MachineName>_<SessionBeanName>_0.<TimeStamp>`. Each (tab-delimited) entry in the file contains the following format:

Summary	Meaning
Timestamp	This timestamp is in a sortable format (ISO 8601 complete) and indicates the time at which the method was invoked. The International Standard for the representation of dates and times is ISO 8601. It displays the timestamp with the accuracy to seconds. The format of the timestamp is YYYYMMDDTHHMMSS. Note that the "T" appears literally in the string, to indicate the beginning of the time element, as specified in ISO 8601.
Machine name	The name of the application server machine on which this function executed.
Session Bean Name	The name of the statistics class, Statistics, is always printed.
Process ID	Currently hard-coded to zero.

Summary	Meaning
Server function signature	The function signature including class and method name, and method argument types.
Success indicator	A flag indicating whether the server function succeeded with no error returned to the client. A value of 1 indicates success, a value of 0 indicates failure. The specific error message is not recorded
Elapsed time in milliseconds	This is the time spent (in milliseconds) executing this function excluding time spent by the middleware software in dispatching the function call and marshaling arguments

Table 8.3 Statistics File Elements

8.7 Localization

In cases where log messages should be localizable, class `Localis-ableString` can be used. See Section 9.9, *Localized Output*. However it is important to note that logged messages are typically targeted at a system administrator who may have a different locale to the current user. For example if the user uses English and the administrator uses French, then the Cúram default locale will be French and the log message should be written in French. In the following example, the default server locale is explicitly passed into `getMessage`, otherwise `getMessage` would return a string corresponding to the users locale rather than the Cúram server locale.

```
import curam.util.resources.ProgramLocale;

// Create a localizable message
curam.util.exception.LocalisableString e =
  new LocalisableString(EXAMPLE.ID_EXAMPLE_MESSAGE);
e.arg(someIdentifier);


// WRONG! This logs the message in the current users locale,
// not that of the Cúram server.
curam.util.resources.Trace.kTopLevelLogger.info(e.getMessage());


// RIGHT: The message is logged using the Cúram server locale.
curam.util.resources.Trace.kTopLevelLogger.info(
  e.getMessage(ProgramLocale.getDefaultServerLocale()));
```

Example 8.6 Localizable logging example in application code

Notes

[1]*log4j* is a logging framework provided by the Apache Jakarta project (see *The complete manual - log4j*, Gulcu).

[2]While *log4j* is designed to impose a minimal overhead it cannot avoid the cost of the parameter construction inside the method invocation. Application developers must take this into consideration.

[3]The set of possible levels (in order of priority) defined by *log4j* are ALL, DEBUG, INFO, WARN, ERROR, FATAL and OFF. Only those items logged at the specified level or higher levels will be included in the log.

Chapter 9

Using Exceptions

9.1 Overview

As the *Java* language provides full support for exceptions, they are the recommended mechanism for handling errors in an *IBM Cúram Social Program Management* application. The advantage of using exceptions to handle errors is that it saves the developer from having to check the status of each operation attempted. A single `try..catch` construct can enclose many statements, each of which could raise an exception.

In a Cúram application, exceptions can originate from various parts of generated code. For example the Database Access Layer (DAL) throws exceptions in the event of a database error, application developers can throw predefined exceptions or customized exceptions. There are two basic forms of exceptions used; checked and unchecked.

Checked exceptions are subclasses of `curam.util.exception.AppException` and `curam.util.exception.InformationalException`. These exceptions must be explicitly caught or listed in the `throws` clause of the method.

Unchecked exceptions are subclasses of `curam.util.exception.AppRuntimeException`. These exceptions do not have to be explicitly handled as they inherit from the *Java* `Exception` and `RuntimeException` classes respectively. Typically, database problems (such as a `RecordLockedException`) are thrown as unchecked exceptions. This means that there is no need for code to tediously check for a `RecordLockedException` each time the database is accessed.

In a Cúram application, checked exceptions can arrive at the Remote Interface Layer (RIL), despite being checked, a `throws` clause can unwind all the way to the RIL. Once here they are converted to a different form of exception which is thrown to the client, and may write information from the

exception to the log file. To avoid this a developer can write code to catch exceptions and handle them and/or re-throw them before the exception reaches the RIL.

The following happens when the RIL catches a checked exception:

- The text for the exception is loaded from a message catalog file.

- If the exception is loggable, then the text will be formatted, with arguments inserted and written to the log file in the default server language.

- If the exception is loggable and includes a stack trace this will be written to the log file.

- An exception is created and thrown to the client. This contains the name of the message catalog, the ID of the message, and the exception arguments if any.

- The client receives the exception and uses the catalog name and message ID to look-up a localized version of the message. It then inserts and formats the arguments into a message and displays the message.

The RIL also catches unchecked exceptions to perform default actions.

- The text for the exception is loaded from a message catalog file.

- The text is formatted with arguments inserted and written to the log file in the default server language.

- A stack trace is written to the log file.

- A new exception is created and thrown to the client. This exception states that the original exception was *Unhandled*. The original exception is mapped because the descriptive text is at too low a level to make sense to a user.

  The newly created exception contains a nested exception which has the details of the original exception - namely the name of the message catalog, the ID of the message, and the exception arguments if any.

  This mapping happens for all but four unchecked exceptions. These exceptions are left untouched because the descriptive text produced is readable to a user. These are `RecordChangedException`. `RecordDeletedException`, `RecordLockedException` and `ReadmultiMaxException`.

- When the client receives the exception and uses the catalog name and message ID to look-up a localized version of the message. It then inserts and formats the arguments into the message and displays the message.

9.2 Constructing an Exception

Exceptions[1] are typically created with a catalog name and message identifier. If these are not specified default values are used. The server infrastruc-

ture will take care of delivering the message text to the client and/or log file.
For example:

```
if (DatabaseFieldIsNull()) {
  curam.util.exception.AppException e = new
      AppException(MAINTENANCE.ID_NULL_INDICATOR);
  throw e;
}

// This can also be written as follows
if (DatabaseFieldIsNull() ) {
  throw new curam.util.exception.AppException
        (INFRASTRUCTURE.ID_NULL_INDICATOR);
}
```

Example 9.1 Constructing an AppException

The purpose of exceptions is to communicate the fact that an error has oc-
curred and to communicate information about that error. Often it is neces-
sary to include additional information as well as the error code. This can be
done using arguments.

Arguments are attached to an exception before it is thrown and are intended
to be ultimately included in the error message displayed at the client and/or
the server log file.

To attach an argument to an exception, the arg method (.arg()) is used.
Example 9.2, *Using the arg method with a primitive type* shows a code ex-
ample of how to use the *arg* method to attach an argument to an exception.

```
// set a status code for the error which occurred
long lngErrorCode = -1;

// create the exception.
curam.util.exception.AppException e = new
        AppException(MAINTENANCE.ID_SYSTEM_ERROR);

// Include this status code with the exception.
e.arg(lngErrorCode);

// now throw the exception
throw e;
```

Example 9.2 Using the arg method with a primitive type

The arg method supports the addition of many different types of arguments
to an exception. Such primitive types include long, boolean or double while
complex types e.g. Date, DateTime, Money and
CodeTableItemIdentifier objects can also be added. See the
JavaDoc for curam.util.exception.AppException for more de-
tails.

```
// Create a codetable identifier to describe domain type.
curam.util.type.CodeTableItemIdentifier aCodeIdentifier =
  new CodeTableItemIdentifier
    (DOMAINTYPE.TABLENAME, DOMAINTYPE.INT32);

// create the exception to flag an invalid data type
curam.util.exception.AppException e = new
  AppException(WORKFLOW.ERR_ANSWER_NOT_VALID_DATATYPE);

// Include the domain type code with the exception.
```

```
e.arg(aCodeIdentifier);

// now throw the exception
throw e;
```

Example 9.3 Using the arg method with a complex type

9.3 Creating Messages with Argument Placeholders

Argument place holders are tokens which are included in the error message source text and are replaced by an argument at runtime.

Place holders are of the form %nc, where n is the argument number (of 1 or more), and c is a single character denoting the argument type as follows:

- s - string

- n - numeric

- d - date

- t - time

- z - date/time

- c - code table item

For example, the source message:

"The first name is %1s and the surname is %2s"

would be displayed as:

"The first name is John and the surname is Smith"

The fact that the place holders are numbered means that they can appear in the message in any order. For example, the source message:

"The second name is %2s and the first name is %1s"

would be displayed as:

"The second name is Smith and the first name is John"

The exception would be constructed and thrown as shown in Example 9.4, *Exception message with argument placeholders*.

```
curam.util.exception.AppException e = new
        AppException(EXAMPLE.ID_EXAMPLE_MESSAGE);
e.arg(Person.FirstName);
e.arg(Person.Surname);
throw e;
```

Example 9.4 Exception message with argument placeholders

9.4 Handling Exceptions

When an exception is thrown in an application, it may be caught within a `try..catch` construct or it may be allowed to filter up to the RIL.

The `try..catch` construct will typically handle the exception in one of the following ways:

- Ignore it and carry on with the next processing step.

  An example of this is where the program must check for the existence of a record on the database. If the DAL throws a `RecordNotFoundException`, then this indicates that the record does not exist. This exception will not be allowed to reach the client, instead it controls how processing is done.

  ```
  bPersonExists = true;
  try {
    dtls = myPerson.read(key);
  }
  catch(RecordNotFoundException rnfe) {
    bPersonExists = false;
  }
  ```

- Pass it upwards to a higher `try..catch` construct by re-throwing the actual exception.

  An example of this is a `try..catch` construct which is interested in only a specific exception. If any other exception is caught then it can be passed on upwards for some other handler to deal with.

  ```
  try {
    myPerson.checkCompleteness(dtls);
  }
  catch(curam.util.exception.AppException e) {
   if(e.equals(APP.ID_INCOMPLETE_DATA)) {
      // set this flag and continue
      bIncompleteData = true;
    } else {
      // do not know how to handle this exception,
      // pass it straight through.
      throw e;
    }
  }
  ```

- Create a new exception and throw the new exception.

  An example of this is where the handler would replace a generated DAL exception with an application exception containing a more user-friendly application-specific error message.

  ```
  catch(RecordNotFoundException rnfe)
  {
    curam.util.exception.AppException e = new
      AppException(APP.NO_SUCH_PERSON);
    // substitute the message for the exception.
    // (The new message includes the ID number of
    // the record we searched for.)
    e.arg(dtls.personIDNumber);
    throw e;
  }
  ```

- Create a new exception, attach the original exception to this new excep-

tion, and raise the new exception.

An exception can be constructed with a pointer to another exception as follows:

```
catch(curam.util.exception.AppException
origException) {
  curam.util.exception.AppException newException = new
          AppException(MYAPP.ID_MYMESG, origException);
  throw newException;
}
```

This has the effect of creating a linked list of exceptions with the most recent exception at the head of the list. This allows a detailed history of an exception to be built up for auditing or debugging purposes.

9.5 Logging Exceptions

Exceptions can be optionally logged to the application log file by setting its loggable flag using the `setLoggable` method.

Loggable exceptions are written to the application log file by the RIL. The exception message is read from the error message catalog file, the exception arguments, if any, are inserted into the text and this parsed text is written to the log file.

An exception is treated as loggable if its loggable flag is set or if the loggable flag is set on any attached exceptions.

If the exception being logged has any other exceptions attached, then these exceptions are also logged.

9.6 General Exception Guidelines

- Follow the processing specification for the method, this should describe the error situations that can be encountered. When actually writing and testing the code, look out for sources of errors that might have been overlooked.

- Do not try to add a "catch-all" for unanticipated errors; the server infrastructure can handle these better than you can. Do not wrap entire operations with error handlers.

- Do handle exceptions where you are in a position to add more specific information about what has happened, such as converting "record not found" into "bank account not found".

- Do gain an understanding of the standard exceptions defined in the *core infrastructure*. Be aware of the types of exceptions that can be thrown by generated database manipulation operations of entity objects:

    - `RecordNotFoundException` can be thrown by singleton reads, updates and removes of the database (entity read, nsread, modify, ns-

modify, remove and nsremove operations). A non standard operation (for example nsmodify and nsremove) will throw this exception irrespective of the uniqueness of the key that is passed into it.

- `RecordNotFoundException` can be thrown by nonkeyed updates and removes of the database (entity nkremove and nkmodify).

- `RecordDeletedException` is always thrown in precedence to a `RecordNotFoundException`.

- `RecordDeletedException` can be thrown when an optimistic update fails because the target record has been deleted. With optimistic locking enabled the record is re-read to obtain the version number. If the record is no longer present this exception is thrown.

- `DuplicateRecordException` can be thrown by insert and update operations (entity insert, nsinsert, modify, nsmodify, nkmodify operations).

- `RecordChangedException` and `RecordDeletedException` can be thrown by update operations with optimistic locking. `RecordDeletedException` is thrown by entities which have optimistic locking enabled in preference to `RecordLockedException`.

- `MultipleRecordException` can be thrown by singleton reads of the database (entity read, nsread, nkread operations) if multiple records are found which meet the specified selection criteria.

- `ReadmultiMaxException` can be thrown by multiple reads of the database (entity readmulti, nsmulti, nkreadmulti operations) if more record are retrieved than the maximum specified in the application model.

- `RecordLockedException` can be thrown by any of the entity operations if a deadlock or lock timeout occurs.

- `OtherDatabaseException` can be thrown by any of the entity operations if the database reports an error which does not map to one of the above exceptions.

9.7 Coding Conventions for Exceptions

- Under normal circumstances don't create your own subclasses of `AppException` or `AppRuntimeException`.

- Use exception chaining and exception logging when handling serious errors (the definition of "serious" is application-specific).

- When writing the text of errors in a message file, be aware of localization issues. Do not write code which simply replaces placeholders with

hard-coded literals as shown in Example 9.5, *Incorrect usage of hard-coded literals.*

```
//Check that BankAccount entity exists:
bankAccountKey.accountNumber = argIn.accountNumber;
try {
  bankAccountDtls = bankAccount.read(bankAccountKey);
} catch (RecordNotFoundException rnf) {
  //This is a SERIOUS error
  curam.util.exception.AppException e = new AppException(
                        COOKBOOK.ID_NO_SUCH_ACCOUNT, rnf);
  e.setLoggable(true);    //make sure it gets logged
  e.arg("not found"); // NOT LOCALIZABLE!!!
  throw e;
}
```

Example 9.5 Incorrect usage of hard-coded literals

9.8 Using Record Not Found Indicator

Each of the singleton reads of the database (entity read, nsread, nkread operations) which could potentially throw a `RecordNotFoundException` has overloads added to take a Record Not Found Indicator variable.

The reasons for providing a Record Not Found Indicator are:

- To save the overhead of creating and throwing an exception whenever a record cannot be found, as this is an expensive process in some *JVM*s.

- To make it easier to write code which simply checks for the existence of a record.

This indicator (`curam.util.type.NotFoundIndicator`) wraps a boolean value which indicates whether the required record could not be found. When this indicator is passed into one of the above read operations, the operation will never throw a `RecordNotFoundException` if the record does not exist but will instead set the boolean flag inside `NotFoundIndicator` to `true`, and return a value of `null`. If the record is found, the boolean flag inside `NotFoundIndicator` is set to `false`, and the record is returned.

Whenever a developer wishes to pass a `NotFoundIndicator` into a singleton read operation, it is always passed in as the first argument. This is shown in the following examples:

```
try {
  bankAccountDtls = bankAccount.read(bankAccountKey);
} catch (RecordNotFoundException rnf) {
  // record was not found...
}
```

Example 9.6 A typical read operation which may throw a

RecordNotFoundException

```
final NotFoundIndicator notFoundInd =
  new curam.util.type.NotFoundIndicator();
bankAccountDtls = bankAccount.read(notFoundInd, bankAccountKey);
if (notFoundInd.isNotFound()) {
  // record was not found...
} else {
  // record was found...
}
```

Example 9.7 The overloaded version of the one above, using the **NotFoundIndicator**

```
try {
  bankAccountDtls = bankAccount.read(bankAccountKey, true);
} catch (RecordNotFoundException rnf) {
  // record was not found...
}
```

Example 9.8 A typical read operation for update which may throw a **RecordNotFoundException**

```
bankAccountDtls =
  bankAccount.read(notFoundInd, bankAccountKey, true);
if (notFoundInd.isNotFound()) {
  // record was not found...
} else {
  // record was found...
}
```

Example 9.9 The overloaded version of the one above, using the **NotFoundIndicator**

9.9 Localized Output

In *IBM Cúram Social Program Management* the client is responsible for converting the text of an exception into the language that a user has chosen. However certain situations do exist where the server must present data to the client for localization. To facilitate these situations the curam.util.exception.LocalisableString class has been introduced. This class is used in a similar manner to AppException as is shown in Example 9.10, *Use of LocalisableString*.

```
curam.util.type.CodeTableItemIdentifier someIdentifier =
  new CodeTableItemIdentifier("someTable", "someCode");
curam.util.exception.LocalisableString e =
  new LocalisableString(EXAMPLE.ID_EXAMPLE_MESSAGE);
e.arg(someIdentifier);
return e.toClientFormattedText();
```

Example 9.10 Use of LocalisableString

This string can be passed back to the client as an output parameter and will be localized by the client.

9.10 Informational Manager

The standard exception handling and string presentation features described in this chapter do not address one scenario. In a number of situations it is useful to present multiple informational messages at one time. For example, during the course of validation a number of warnings, or errors, may occur independently as they are based on different elements of the user input. These should be reported together to simplify the corrective actions that a user must take. The `InformationalManager` class allows for exceptions and informationals to be grouped together in this manner. Example 9.11, *Use of the Informational Manager* shows the use of this class to group informational messages for presentation:

```
import curam.util.exception.InformationalElement;
import curam.util.exception.InformationalException;
import curam.util.exception.InformationalManager;
import curam.util.exception.LocalisableString;
import curam.util.internal.security.struct.LoginMessage;
import curam.util.internal.security.struct.LoginMessageList;
import curam.util.message.INFRASTRUCTURE;
import curam.util.resources.GeneralConstants;

class InformationalManagerDemo {

  public LoginMessageList checkLoginStatus()
  throws InformationalException {

    // Create an informational manager to store the
    // results of the validation checks. A transaction wide
    // version can be obtained via
    // TransactionInfo.getInformationalManager().
    final InformationalManager informationalManager =
      new InformationalManager();

    // Informational #1
    // Create an informational string for presentation to
    // the client: this specifies the password will expire
    // in 6 days
    LocalisableString infoMessage1 = new LocalisableString(
        INFRASTRUCTURE.INFO_ID_PASSWORD_EXPIRING);
    infoMessage1.arg(6);
    // Add this informational string to the informational
    // manager
    informationalManager.addInformationalMsg(infoMessage1,
        GeneralConstants.kEmpty,
        InformationalElement.InformationalType.kWarning);

    // Informational #2
    // Create an informational string for presentation to
    // the client: this specifies the user will be locked
    // out if they do not change their password in the next
    // 10 logins.
    LocalisableString infoMessage2 = new LocalisableString(
        INFRASTRUCTURE.INFO_ID_LOG_ATTEMPTS_EXPIRING);
    infoMessage1.arg(10);
    // Add this informational string to the informational
    // manager
    informationalManager.addInformationalMsg(infoMessage2,
        GeneralConstants.kEmpty,
        InformationalElement.InformationalType.kWarning);

    // The informationals must now be converted to a format
    // suitable for return to the client.
    final String[] informationalArray = informationalManager
```

```
      .obtainInformationalAsString();
  // The array of informational strings must be
  // transferred to an array of structs because we
  // cannot return an array of strings directly. Each
  // string goes into one struct (LoginMessage) and
  // this is aggregated into a list by struct
  // LoginMessageList.
  // LoginMessage : A struct containing one string
  // named 'message'.
  // LoginMessageList : A struct which aggregates
  // LoginMessage as member 'dtls'.
  final LoginMessageList result = new LoginMessageList();
  for (int i = 0; i != informationalArray.length; i++) {
    LoginMessage warning = new LoginMessage();
    warning.message = informationalArray[i];
    result.dtls.addRef(warning);
  }
  return result;
}

}
```

Example 9.11 Use of the Informational Manager

There are a number of points worth emphasizing in this code fragment:

- This sample is based around the presentation of informationals to the client. It does not throw an exception, and therefore it is a successful invocation of the method. This means the transaction will be committed and any database updates will be made permanent. It is the responsibility of the client screen for this sample to handle the return value of the operation as a collection of informationals.

- `InformationalManager.failOperation()` can be used to fail the invocation depending on whether or not the informational manager contains any warnings or errors. If the informational manager contains an error or warning then this method will throw an exception which means the transaction will be rolled-back. Otherwise this method does nothing and the transaction is allowed to continue. The full details of this operation are described in the API documentation (JavaDoc) shipped with *IBM Cúram Social Program Management*.

- The second parameter to `InformationalManager.addInformationalMsg` currently populated with GeneralConstants.kEmpty (as in Example 9.11, *Use of the Informational Manager*) is intended to name a field. However, this is not supported in the current release

The *Cúram Web Client Reference Manual* should be consulted to determine the client side configuration that is necessary to use the `InformationalManager`; at its simplest the field in the struct containing the informationals must be named in the UIM.

The `InformationalManager` logs informationals to the Curam log. Please see Chapter 8, *Logging* for details on Logging.The informationals are logged in the following way:

- Logging of the informationals is only performed at the time when they

are added to the `InformationalManager` (i.e. when calling `InformationalManager.addInformationalMsg()`).

- Fatal errors and errors are logged at the top level logger using the error level.

- Warnings are logged at the top level logger using the info level.

Notes

[1]The following sections focus on use of `AppException` rather than `AppRuntimeException` as this is typical of production code. However, `AppRuntimeException` can be created and manipulated in the same way.

Chapter 10

Message and Code Table Files

10.1 Overview

This chapter describes message catalog and code table files and how they are used in the *IBM Cúram Social Program Management* application. Cúram message catalogs allow an application to be localized without manipulating hand-crafted code, while Cúram code table files allow an application to use a level of indirection when storing commonly used constants on the database, e.g., Ms., Mr. This chapter introduces message and code table fundamentals, and explains how they can be augmented to produce customized messages and code tables in a Cúram application.

10.2 Message Files

Traditionally message files or catalogs are binary files used for holding text messages associated with an application. Each message catalog had a one-to-one association with a symbol definition file. The symbol definition file was examined at compile time and the message catalog at run-time. Using this form of indirection allows an application to be localized without a re-compilation being necessary.

In keeping with this approach, Cúram message catalogs are generated from message `.xml` files using a command-line build utility called **msggen** (**build msggen**). Generating from a message `.xml` file produces two outputs: a message catalog file (one *Java* `.properties` file is generated for each locale specified) and a symbol definition file (a standard *Java* class file). The symbol definition file is a *Java* file containing constants (in *Java* terms, a constant is a `static final`) for message identifiers enumerated in the message.xml file, and the name of the message file itself. This file should be imported into any *Java* source files which use that catalog. The message catalog is a properties file opened by the Cúram application at runtime.

The **msggen** build target performs the merge of message files and then translates the resultant message file (which are stored in `/build/svr/message/scp`) into symbol definition (*Java* code) and message catalog (property) files.

msggen is automatically invoked by the provided build scripts, against those message files which are placed in the suggested source locations, i.e., the `/message` directory of a component.

10.2.1 The Format of Message Files

The message `.xml` file is an XML document which is made up of a number of distinct elements combined with the core message elements; see Example 10.1, *Example of Message text file*.

As a standard XML document, the `encoding` attributed indicates that the file is encoded in UTF-8. It should be noted that this encoding will be respected and maintained by an XML aware editor. However, other editors (such as *TextPad*) do not maintain this encoding by default. A file which contains UTF-8 characters may have to be specifically saved as UTF-8 with these editors.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A sample message file. -->
<messages package="curam.message">
  <message name="ERR_XRV_EXISTING_OVERLAP">
    <locale country="US" language="en">
      More than 1 overlapping Assessment has been found.
    </locale>
  </message>
    <message name="ERR_CREATION_DATE_EMPTY">
    <locale country="US" language="en">
      You must enter a creation date
    </locale>
  </message>
</messages>
```

Example 10.1 Example of Message text file

The following sections detail the `message.xml` file elements and attributes.

The `<messages>` Element

The `<messages>` element is the root element of a message file, and it groups all other elements together. The `messages` element has the following attribute:

Attribute Name	Required	Default	Description
package	Yes	None	The *Java* package name to use for the generated *Java* file.

Table 10.1 Attributes of the messages Element

The **`<message>`** Element

The `<message>` element groups a number of `<locale>` elements together. The `message` element has the following attributes:

Attribute Name	Re- quired	Default	Description
name	Yes	None	Uniquely identifies the message.
removed	No	false	Set to `true` to indicate if the message is to be removed and hence not included in the generated artefacts.

Table 10.2 Attributes of the message Element

The **`<locale>`** Element

The `<locale>` element details the text of the message for one of the supported locales. Since the message files are XML, it is not necessary to use *Java* escape characters. Special characters can be inserted by using the XML entity references in the message files. These will be converted to the actual characters in the properties file. For example ¢ and $ will result in the cent and dollar symbols, respectively, being put in the properties file. Care must be taken to only specify characters that can be supported by the target properties file on your platform and for your operating system locale.

The `locale` element has the following attributes:

Attribute Name	Re- quired	Default	Description
language	Yes	None	To be included during generation of the message artefacts each `<locale>` element must specify a language (and optional country) attribute that corresponds to a supported locale. The SERV- ER_LOCALE_LIST environment variable is a comma separated list defining the set of locales that are supported, where the locale is either simply `language` or `lan- guage_country`. For example: SERVER_LOCALE_LIST=en, en_US, en_GB.
country	No	None	Set to the country relevant to the loc- ale `language` attribute.

Table 10.3 Attributes of the locale Element

10.2.2 Customizing a Message File

Message text files are located in the `/message` directory of a component. The Social Program Management Platform is shipped with a set of message files. These may be overridden by placing new message files in the `SERVER_DIR/components/<custom>/message` directory, where *<custom>* is any new directory created under components that conform to the same directory structure as `components/core`. This mechanism avoids the need to make changes directly to the out-of-the-box application, which would complicate later upgrades.

> ℹ **Note**
>
> If the `package` attribute in the overridden message file is modified, then the customization will not work.

This override process involves merging all message files of the same name according to a precedence order. The order is based on the `SERVER_COMPONENT_ORDER` environment variable. This environment variable contains a comma separated list of component names: the left most has the highest priority, and the right most the lowest.

```
SERVER_COMPONENT_ORDER=custom,Appeal,ISProduct,sample
```

Example 10.2 SERVER_COMPONENT_ORDER example

The order in Example 10.2, *SERVER_COMPONENT_ORDER example*, shows that the precedence of `Appeal` is higher than that of the `sample` component. The `core` component always has the lowest priority and as such does not need to be specified. Any components that are not specified are placed alphabetically above `core` and below those that are specified.

> ℹ **Note**
>
> After changing the component precedence order in `SERVER_COMPONENT_ORDER` it is necessary to perform a clean build to ensure that you are using the appropriate files. This is done by invoking **build clean server**.

When merging message files, the components listed in the `SERVER_COMPONENT_ORDER` are taken in order of highest to lowest priority. In Example 10.2, *SERVER_COMPONENT_ORDER example* message files from the `sample` component are merged with the message files located in the `core` component. The message files from `ISProduct` are then merged into the intermediate results and the merge process continues until the messages in the `custom` component are merged.

Rules of Message Merges

Message files are merged based on precedence order. As described above there is always a more important main/source message file, and a file which is being merged into it. The second file is called the merge file in the following sections.

The merging rules described below are applied to decide if the `<message>` and `<locale>` elements should be merged into the new message file.

- A `<message>` will be merged into a new message file if the `<message>` is not already present in the new file.

- A `<locale>` will be merged into a named `<message>` element in the new message file if the `<locale>` is not already present in the `<message>` of the new message file.

Duplicate messages will always be overwritten by the message file in the component with the highest precedence order. The main message file of Example 10.3, *Sample main message file*, and the merge file of Example 10.4, *Sample merge message file*, illustrate these rules:

```
<messages package="curam.message">
  <message name="ERR_SAMPLE_VALIDATION_MSG">
    <locale country="US" language="en">
      The specified color is not valid.
    </locale>
  </message>
  <message name="ERR_SAMPLE_ERROR_MSG">
    <locale country="US" language="en">
      An external resource is not available.
    </locale>
  </message>
</messages>
```

Example 10.3 Sample main message file

```
<messages package="curam.message">
  <message name="ERR_SAMPLE_VALIDATION_MSG">
    <locale country="GB" language="en">
     The specified colour is not valid.
    </locale>
  </message>
   <message name="ERR_SAMPLE_NEW_MSG">
     <locale country="GB" language="en">
       An example of localisation.
     </locale>
   </message>
   <message name="ERR_SAMPLE_REMOVED_MSG" removed="true">
     <locale language="en">
      This message will be removed.
     </locale>
  </message>
</messages>
```

Example 10.4 Sample merge message file

As a result of the merge process the new message file produced would be:

```
<messages package="curam.message">
  <message name="ERR_SAMPLE_VALIDATION_MSG">
    <locale country="GB" language="en">
      The specified colour is not valid.
```

```
      </locale>
      <locale country="US" language="en">
        The specified color is not valid.
      </locale>
    </message>
    <message name="ERR_SAMPLE_ERROR_MSG">
      <locale country="US" language="en">
        An external resource is not available.
      </locale>
    </message>
    <message name="ERR_SAMPLE_NEW_MSG">
      <locale country="GB" language="en">
        An example of localisation.</locale>
      </locale>
    </message>
    <message name="ERR_SAMPLE_REMOVED_MSG" removed="true">
      <locale language="en">
        This message will be removed.
      </locale>
    </message>
  </message>
</messages>
```

Example 10.5 Resulting Message File

10.2.3 Artefacts Produced by msggen Build Target

The *Java* artefacts (symbol definition and message catalog files) produced from a merged message file, are placed in the `/build/svr/message/gen/<package>` directory, where `<package>` is the package attribute specified in the message file. For example, *package="curam.message"* would result in the *Java* artefacts being placed in the `/build/svr/message/gen/curam/message` directory.

The directory contains the *Java* files (which are locale independent) and the property files (which are locale dependent) which are named `<Message File name>_<specific language>_<specific country>.properties`.

> ℹ️ Note
>
> If message files of the same name exist in different components with a different `package` attribute value, then the generated artefacts (symbol definition and message catalog files) produced are placed in the package specified by the message file of the component with the highest precedence order (as listed in the `SERVER_COMPONENT_ORDER` environment variable).

These artefacts are best illustrated by example:

```
package curam.message;
import curam.util.message.CatEntry;
import curam.util.message.MessageCatalog
public final class SampleMessages {

  private static final MessageCatalog kCat =
    new MessageCatalog("curam.message.SampleMessages");

  /**
   * BpoActivity:ERR_SAMPLE_VALIDATION_MSG
   * en_UK = The specified colour is not valid.
```

```
   * en_US = The specified color is not valid.
   */
 public static final CatEntry ERR_SAMPLE_VALIDATION_MSG

   = kCat.entry("ERR_SAMPLE_VALIDATION_MSG");

 /**
  * BpoActivity:ERR_SAMPLE_ERROR_MSG
  * en_US = An external resource is not available.
  */
 public static final CatEntry ERR_SAMPLE_ERROR_MSG
   = kCat.entry("ERR_SAMPLE_ERROR_MSG");

 /**
  * BpoActivity:ERR_SAMPLE_NEW_MSG
  * en_GB = An example of localisation.
  */
 public static final CatEntry ERR_SAMPLE_NEW_MSG
   = kCat.entry("ERR_SAMPLE_NEW_MSG");
}
```

Example 10.6 Java file produced from merged message file

```
ERR_SAMPLE_VALIDATION_MSG=The specified colour is not valid.
ERR_SAMPLE_NEW_MSG=An example of localisation.
```

Example 10.7 Sample (UK) Properties produced from message file

At the end of the `msggen` step these property files are placed into a `.jar` file which is used by the client to localize the messages that are returned to it.

10.2.4 Retrieving Messages from Message Files

A message file can contain any number of locales for a named message, and as a result a mechanism needs to be in place to return the correctly localized message for a running instance of Cúram. Messages are retrieved from a message file based on the locale property which includes a language and, optionally, a country. The message file look up returns a matching localized message for a named message identifier. For example, if the runtime locale is set to `en_US` where "en" is the language and "US" is the country, a message look up for the message named A_MESSAGE (Example 10.8, *Message File Search*) will return the text "The message". If however the runtime locale was set to "fr" the text "Le message" would be returned.

```
<messages package="curam.message">
<message name="A MESSAGE">
  <locale country="US" language="en">The message</locale>
  <locale language="fr">Le message</locale>
</message></messages>
```

Example 10.8 Message File Search

Since message files are not guaranteed to contain an entry for each message that matches the runtime locale, a fall back mechanism is in place to guarantee that if possible a localized message is returned when a look up is per-

formed. Once a message of a given name has been found, and there is no direct match with the specified locale, the rules for fall back are as follows:

- If the runtime locale is set to include a language and country, the country is removed and the search looks for a matching language only. Looking up the message named A_MESSAGE in Example 10.9, *Message File FallBack One* with runtime locale en_US will return the message text "The message".

```
<messages package="curam.message">
  <message name="A_MESSAGE">
    <locale language="en">The message</locale>
  </message>
</messages>
```

Example 10.9 Message File FallBack One

The underlying message lookup mechanism is provided by the JDK class java.util.ResourceBundle. Please refer to the relevant JDK JavaDoc for details of this classes functionality and further details of the fallback mechanism provided.

There is no default behavior for message file lookup. If the runtime locale does not find a match in the message file after applying the fall back rules described by java.util.ResourceBundle, a MissingResourceException is returned and server logs are updated if appropriate.

10.2.5 Writing Messages To Server Logs

Messages from message catalogs are used in many instances in Cúram and localized at runtime as described in Section 10.2.4, *Retrieving Messages from Message Files*. Localization of server log messages is different in that it is performed by the server infrastructure based on the default server locale. In this case, the locale used when writing to Cúram server logs is set by configuring the curam.environment.default.locale property in Application.prx.

10.2.6 Localizing SDEJ Message Files

It is possible to localize or modify the message files shipped with the Cúram SDEJ. These message files are located in the message directory of the SDEJ and are in the same format as Cúram application message files but with the extension .iml.

To localize these files copy the particular .iml message file to be modified from the SDEJ to the message directory of a component in your Cúram application, for example, SERVER_DIR/components/custom/message. The .iml message file can then be modified in the same way as any message file, overriding a message or adding a new locale for all the messages.

ℹ️ **Note**

If the `package` attribute in the message file is modified the localization will not work.

The **msggen** target, when run, will merge the localized `.iml` message file with the original one located in the SDEJ. The localized message file will have the higher precedence order. It will then generate the properties files only and include them in the `messages.jar` file created. The `messages.jar` file will always be on the classpath before the default SDEJ messages in a runtime application.

10.3 Code Table Files

Code table files allow a Cúram application to use a level of indirection when storing commonly used constants on the database. Like message files, code table files are shipped with Cúram and can be customized by adding new code table files to new components in the SERVER_DIR/components/<custom>/codetable directory, where *<custom>* is any new directory created under `components` that conforms to the same directory structure as `components/core`. Code table files can contain one code table or a number of code tables that are linked as a hierarchy.

Generating code tables produces two outputs: a code table SQL file to place the codes on the database, and a symbol definition file (a standard *Java* class file). The symbol definition file is a *Java* file containing constants for code table identifiers used in the code table XML file. The generation of code table hierarchies also produces .properties files as described in Section 10.3.3, *Artefacts Produced by ctgen Build Target*.Generating code tables is supported by the build target **ctgen**.

For more information on code tables also consult the Domain Definitions chapter in the *Cúram Modeling Reference Guide* and the *Cúram Web Client Reference Manual*.

10.3.1 The Format of Code Table Files

The code table file is an XML document which is made up of a number of distinct elements. Example 10.10, *Sample Main Code Table File 1*, shows a sample code table.

As a standard XML document, the `encoding` attributed indicates that the file is encoded in UTF-8. It should be noted that this encoding will be respected and maintained by an XML aware editor. However, other editors (such as *TextPad*) do not maintain this encoding by default. A file which contains UTF-8 characters may have to be specifically saved as UTF-8 with these editors.

The following sections detail the elements and attributes of a code table file.

The `<codetables>` Element

The `<codetables>` element is the root element of a code table file and it groups all other elements together. The `codetables` element has the following attributes:

Attribute Name	Required	Default	Description
package	Yes	None	Specifies the package the generated symbol definition *Java* file will be part of.
hierarchy_name	No	None	Identifies the code table file as containing a hierarchy of code tables.

Table 10.4 Attributes of the codetables Element

The **<description>** Element

The `<description>` element is an optional sub-element below the `<codetables>` root. It is used to define a description for the code tables. It should be listed first, before the other sub-element, `<codetable>`, of `<codetables>`. It should only be listed once. There are no attributes for the `description` element.

The **<codetable>** Element

The `<codetable>` element is a sub-element below the `<codetables>` root. The `<codetable>` element should follow the `<description>` element, if it's specified. For an ordinary code table file definition only a single `<codetable>` element can be defined. If a `hierarchy_name` attribute has been specified in the `<codetables>` multiple `<codetable>` elements are allowed as long as they are linked correctly in a hierarchy.

The `codetable` element groups a number of `<code>` elements together and an optional `<codetabledata>` element.

The `<codetable>` element has the following attributes:

Attribute Name	Required	Default	Description
name	Yes	None	A unique identifier for the code table. The `name` attribute is trimmed of leading and trailing spaces on code table generation. Some restrictions apply to the `name` attribute when the `<displaynames>` element is specified. Please see Section 10.3.3, *Artefacts Pro-*

Attribute Name	Re-quired	Default	Description
			duced by ctgen Build Target for further details.
`java_identifier`	Yes	None	The name of the generated symbol definition *Java* file.
`parent_codetable`	No	None	Used to define the name of the parent code table in the hierarchy, where the code table file has been defined as a hierarchy of code tables.

Table 10.5 Attributes of the codetable Element

The **`<codetabledata>`** Element

The `<codetabledata>` element is an optional sub-element of `<codetable>` that groups the locale-specific comments for a codetable. Each `<codetable>` element can have one optional `<codetabledata>` element. The `<codetabledata>` element can contain multiple optional `<locale>` elements.

> ℹ **Note**
>
> The `<codetabledata>` element and its child elements are optional elements.

The `<codetabledata>` element has the following attributes:

Attribute Name	Re-quired	Default	Description
`language`	Yes	None	Specifies the language portion of the locale for the `codetabledata` element.
`country`	No	None	Specifies the country portion of the locale for the `codetabledata` element.

Table 10.6 Attributes of the codetabledata Element

The **`<locale>`** Element

The optional `<locale>` element can occur multiple times for the `<codetabledata>` element. Each `<locale>` element can contain one optional `<comments>` element.

The `locale` element has the following attributes:

Attribute Name	Re-quired	Default	Description
language	Yes	None	Specify a language that corresponds to a supported locale.
country	No	None	Specify a country that corresponds to a supported locale and language.

Table 10.7 Attributes of the locale Element

The **<comments>** Element

The optional <comments> element is used to store the locale-specific comments for a code table.

The comments element has no attributes.

The **<displaynames>** Element

The <displaynames> element groups a number of code table hierarchy <name> elements together. It is an optional element. However, if present it must contain at least one <name> element. The displaynames element has no attributes.

The **<name>** Element

The <name> element is required when the <displaynames> element is present. When displaying the <name> values on the client, the name that contains the locale for the current user is displayed. However, if the current user's locale does not match any of the locales specified within the <name> element, then the <codetable> name attribute is displayed.

The name element has the following attributes:

Attribute Name	Re-quired	Default	Description
language	Yes	None	Specifies the language portion of the locale for the name element.
country	No	None	Specifies the country portion of the local for the name element.

Table 10.8 Attributes of the name Element

The **<code>** Element

The <code> element is a sub-element of <codetable> and groups a number of <locale> elements together. The code element has the following attributes:

Attribute Name	Re-quired	Default	Description
`value`	Yes	None	A unique identifier for the code in the code table.
`status`	Yes	None	Indicates if the code table is enabled and selectable in the list of codes as displayed on the client. It can be set to either `ENABLED` or `DISABLED` and if set to anything else it is considered to be `DISABLED`.
`default`	No	None	Indicates if this is the default code for the code table. There should only ever be one default specified. The default code is used to define the initially selected value in an editable code table list in the client. For more information consult the `Cúram Web Client Reference Manual`.
`java_identifier`	No	None	Used as part of the generated symbol definition *Java* file
`removed`	No	`false`	Set to `true` to indicate if the code is to be removed and hence not included in the generated artefacts
`parent_code`	No	None	Used to define the name of the code in the specified parent code table in the hierarchy that this code is linked to. See Section 10.3.4, *Code Table Hierarchy* for more information on defining a code table hierarchy.

Table 10.9 Attributes of the code Element

The **`<locale>`** Element

The `<locale>` element contains two mandatory sub-elements (`<description>` and `<annotation>`) and one optional sub-element `<comments>`, which are used to describe the code.

To be included during generation of the code table artefacts, each `<locale>` element must specify a language (and optional country) attribute that corresponds to a supported locale. The `SERVER_LOCALE_LIST` environment variable is a comma separated list of locales that are supported, where the locale is either simply of the form `language` or `language_country` as shown in this example:

```
SERVER_LOCALE_LIST=en, en_US, en_GB
```

The `locale` element has the following attributes:

Attribute Name	Re-quired	Default	Description
`language`	Yes	None	Specifies a language that corresponds to a supported locale.
`country`	No	None	Specifies a country that corresponds to a supported locale and language.
`sort_order`	No	None	Specifies the order in which the codes in a code table will be displayed in the drop-down list on an edit page in the client.

Table 10.10 Attributes of the locale Element

The **`<description>`** Element

The `<description>` element is used to provide a description for the code. The `description` element has no attributes.

The **`<annotation>`** Element

The `<annotation>` element is used to provide an annotation to the code. The `annotation` element has no attributes.

The **`<comments>`** Element

The optional `<comments>` element is used to store the locale-specific comments for a code table item. This element can be used to provide localized information to aid in understanding the usage for a code table item, and any implication of change to it.

The `comments` element has no attributes.

10.3.2 Customizing a Code Table File

Code table files are located in the `/codetable` directory of a component. The Social Program Management Platform is shipped with a set of code table files. These may be overridden by placing new code table files in the `SERVER_DIR/components/<custom>/codetable` directory, where `<custom>` is any new directory created under `components` that conforms to the same directory structure as `components/core`.

This mechanism avoids the need to make changes directly to the out-of-the-box application, which would complicate later upgrades. Typically code table files are customized to add new entries, localize descriptions or to add new locales.

This override process involves merging all code table files of the same name according to a precedence order. The order is based on the `SERVER_COMPONENT_ORDER` environment variable which contains a comma-separated list of component names: the left most has the highest priority,

and the right most the lowest[1]

Rules of Code Table Merges

Code table files are merged based on precedence order. There is always a more important main/source code table file, and a file which is being merged into it. The second file is called the merge file in the following sections.

The merging rules described below are applied to decide if the `<code>`, `<locale>`, `<displaynames>`, and `<name>` elements should be merged into the new code table file.

- A `<code>` will be merged into a new code table file if its associated `<codetable>` is present in the new file and its `value` attribute is not already present in the new file.

- The `<codetabledata>` element is merged into the `<codetabledata>` element in the new code table file if the `<locale>` element is not already present in the `<codetabledata>` element of the new code table. The `<codetabledata>` element is added into the new code table file even if the `<codetabledata>` is not already present in the new code table file.

- A `<locale>` will be merged into a named `<code>` element in the new code table file if the `<locale>` is not already present in the `<code>` of the new code table.

- A `<displaynames>` element will be merged into a new code table file if its associated `<codetable>` is present in the new file and it is not already present in the new file.

- If the `<displaynames>` element is already present in the new file, then the `<name>` elements will need to be merged. If the `<name>` element with its `language` and `country` attributes is not already present in the new file, then it will be merged into the new file.

The main code table file of Example 10.10, *Sample Main Code Table File 1*, and the merge code table file of Example 10.11, *Sample Merge Code Table File 1*, illustrate the rules of merging `<code>`, `<codetabledata>` and `<locale>` elements.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<codetables package="curam.codetable">
  <codetable java_identifier="ACCEPTANCESTATUS"
             name="AcceptanceStatus">
    <code default="true" java_identifier="ACCEPTED"
          status="ENABLED" value="ACS1">
      <locale language="en" country="US" sort_order="0">
        <description>Accepted</description>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="PROVISIONAL"
          status="ENABLED" value="ACS2">
      <locale language="en" country="US" sort_order="0">
        <description>Provisional</description>
```

```
        <comments>Comments for PROVISIONAL in EN_US</comments>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="REJECTED"
         status="ENABLED" value="ACS3">
      <locale language="en" country="US" sort_order="0">
        <description>Rejected</description>
        <comments>Comments for Rejected in EN_US</comments>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="REMOVED" removed="true"
         status="ENABLED" value="ACS3">
      <locale language="en" country="US" sort_order="0">
        <description>Removed</description>
        <annotation>This message will be removed</annotation>
      </locale>
    </code>
    <codetabledata>
      <locale language="en">
        <comments>Code table comments for
        Country in EN.</comments>
      </locale>
      <locale> language="en" country="US">
        <comments>Code table comments for
        Country in US.</comments>
      </locale>
    </codetabledata>
  </codetable>
</codetables>
```

Example 10.10 Sample Main Code Table File 1

```
<?xml version="1.0" encoding="UTF-8"?>
<codetables package="curam.codetable">
  <codetable java_identifier="ACCEPTANCESTATUS"
            name="AcceptanceStatus">
    <code default="true" java_identifier="ACCEPTED"
         status="ENABLED" value="ACS1">
      <locale language="en" country="GB" sort_order="0">
        <description>Passed</description>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="PROVISIONAL"
         status="ENABLED" value="ACS2">
      <locale language="en" country="GB" sort_order="0">
        <description>Pending</description>
        <comments>Comments for PROVISIONAL in EN_GB</comments>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="REJECTED"
         status="ENABLED" value="ACS3">
      <locale language="en" country="GB" sort_order="0">
        <description>Failed</description>
        <comments>Comments for REJECTED in EN_GB</comments>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="UNKNOWN"
         status="ENABLED" value="ACS4">
      <locale language="en" sort_order="0">
        <description>Unknown</description>
        <annotation></annotation>
      </locale>
    </code>
    <codetabledata>
      <locale language="en">
```

```
        <comments>Code table comments for
        Country in EN.</comments>
      </locale>
      <locale language="en" country="GB">
        <comments>Code table comments for
        Country in GB.</comments>
      </locale>
    </codetabledata>



  </codetable>
</codetables>
```

Example 10.11 Sample Merge Code Table File 1

As a result of the merge process the resulting code table file would be:

```
<codetables package="curam.codetable">
 <codetable java_identifier="ACCEPTANCESTATUS"
     name="AcceptanceStatus">
    <code default="true" java_identifier="ACCEPTED"
         status="ENABLED" value="ACS1">
      <locale language="en" country="US" sort_order="0">
        <description>Accepted</description>
        <annotation></annotation>
      </locale>
      <locale language="en" country="GB" sort_order="0">
        <description>Passed</description>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="PROVISIONAL"
         status="ENABLED" value="ACS2">
      <locale language="en" country="US" sort_order="0">
        <description>Provisional</description>
        <comments>Comments for PROVISIONAL in EN_US</comments>
        <annotation></annotation>
      </locale>
      <locale language="en" country="GB" sort_order="0">
        <description>Pending</description>
        <comments>Comments for PROVISIONAL in EN_GB</comments>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="REJECTED"
         status="ENABLED" value="ACS3">
      <locale language="en" country="US" sort_order="0">
        <description>Rejected</description>
        <comments>Comments for REJECTED in EN_US</comments>
        <annotation></annotation>
      </locale>
      <locale language="en" country="GB" sort_order="0">
        <description>Failed</description>
        <comments>Comments for REJECTED in EN_GB</comments>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="UNKNOWN"
         status="ENABLED" value="ACS4">
      <locale language="en" sort_order="0">
        <description>Unknown</description>
        <annotation></annotation>
      </locale>
    </code>
    <code default="false" java_identifier="REMOVED" removed="true"
         status="ENABLED" value="ACS3">
      <locale language="en" country="US" sort_order="0">
        <description>Removed</description>
        <annotation>This message will be removed</annotation>
```

```
      </locale>
    </code>
    <codetabledata>
      <locale language="en">
        <comments>Code table comments for
        Country in EN.</comments>
      </locale>
      <locale> language="en" country="US">
        <comments>Code table comments for
        Country in US.</comments>
      </locale>
      <locale language="en" country="GB">
        <comments>Code table comments for
        Country in GB.</comments>
      </locale>
    </codetabledata>

  </codetable>
</codetables>
```

Example 10.12 Resulting Code Table File 1

The main code table file of Example 10.13, *Sample Main Code Table File 2*, and the merge code table file of Example 10.14, *Sample Merge Code Table File 2*, illustrate the rules of merging <displaynames> and <name> elements.

```
<codetables
  hierarchy_name="CarHierarchy"
  package="curam.codetable">
  <codetable java_identifier="CarMake" name="CarMake">
    <displaynames>
      <name country="GB" language="en">Car Make CustomGB</name>
      <name language="lt">Masinos Gamintojas</name>
      <name language="en">Car Make Custom</name>
    </displaynames>
    <code default="false" java_identifier="MITS"
     status="ENABLED" value="CMK1">
      <locale language="en" sort_order="0">
        <description>Mitsubishi</description>
        <annotation/>
      </locale>
    </code>
    <code default="false" java_identifier="AUDI"
      status="ENABLED" value="CMK2">
      <locale language="en" sort_order="0">
        <description>Audi</description>
        <annotation/>
      </locale>
    </code>
  </codetable>
  <codetable java_identifier="CarModel" name="CarModel"
    parent_codetable="CarMake">
    <code default="false" java_identifier="COLT"
     parent_code="CMK1" status="ENABLED" value="CML1">
      <locale language="en" sort_order="0">
        <description>Colt</description>
        <annotation/>
      </locale>
    </code>
    <code default="false" java_identifier="LANCER"
     parent_code="CMK1" status="ENABLED" value="CML2">
      <locale language="en" sort_order="0">
        <description>Lancer</description>
        <annotation/>
      </locale>
    </code>
  </codetable>
```

```
</codetables>
```

Example 10.13 Sample Main Code Table File 2

```
<codetables
  hierarchy_name="CarHierarchy"
  package="curam.codetable"
>
  <codetable java_identifier="CarMake" name="CarMake">
    <displaynames>
      <name country="US" language="en">Car Make US</name>
      <name language="fr">Marque</name>
      <name language="en">Car Make Core</name>
      <name language="en" country="GB">Car Make CoreGB</name>
    </displaynames>
    <code default="false" java_identifier="MITS"
      status="ENABLED" value="CMK1">
      <locale language="en" sort_order="0">
        <description>Mitsubishi</description>
        <annotation/>
      </locale>
    </code>
    <code default="false" java_identifier="AUDI"
      status="ENABLED" value="CMK2">
      <locale language="en" sort_order="0">
        <description>Audi</description>
        <annotation/>
      </locale>
    </code>
  </codetable>
  <codetable java_identifier="CarModel" name="CarModel"
    parent_codetable="CarMake">
   <displaynames>
      <name language="en">Car Model</name>
    </displaynames>
    <code default="false" java_identifier="COLT"
     parent_code="CMK1" status="ENABLED" value="CML1">
      <locale language="en" sort_order="0">
        <description>Colt</description>
        <annotation/>
      </locale>
    </code>
    <code default="false" java_identifier="LANCER"
    parent_code="CMK1" status="ENABLED" value="CML2">
      <locale language="en" sort_order="0">
        <description>Lancer</description>
        <annotation/>
      </locale>
    </code>
  </codetable>
</codetables>
```

Example 10.14 Sample Merge Code Table File 2

As a result of the merge process, the resulting code table file would be:

```
<codetables
  hierarchy_name="CarHierarchy"
  package="curam.codetable">
  <codetable java_identifier="CarMake" name="CarMake">
    <displaynames>
      <name country="GB" language="en">Car Make CustomGB</name>
      <name language="lt">Masinos Gamintojas</name>
      <name language="en">Car Make Custom</name>
      <name country="US" language="en">Car Make US</name>
      <name language="fr">Marque</name>
    </displaynames>
    <code default="false" java_identifier="MITS"
```

```
      status="ENABLED" value="CMK1">
      <locale language="en" sort_order="0">
        <description>Mitsubishi</description>
        <annotation/>
      </locale>
    </code>
    <code default="false" java_identifier="AUDI"
     status="ENABLED" value="CMK2">
      <locale language="en" sort_order="0">
        <description>Audi</description>
        <annotation/>
      </locale>
    </code>
  </codetable>
  <codetable java_identifier="CarModel" name="CarModel"
   parent_codetable="CarMake">
    <displaynames>
        <name language="en">Car Model</name>
    </displaynames>
    <code default="false" java_identifier="COLT"
    parent_code="CMK1" status="ENABLED" value="CML1">
      <locale language="en" sort_order="0">
        <description>Colt</description>
        <annotation/>
      </locale>
    </code>
    <code default="false" java_identifier="LANCER"
     parent_code="CMK1" status="ENABLED" value="CML2">
      <locale language="en" sort_order="0">
        <description>Lancer</description>
        <annotation/>
      </locale>
    </code>
  </codetable>
</codetables>
```

Example 10.15 Resulting Code Table File 2

10.3.3 Artefacts Produced by ctgen Build Target

The artefacts produced from the code table file of Example 10.11, *Sample Merge Code Table File 1*, are a symbol definition file (*Java* class) and an SQL file.

The symbol definition file is a *Java* file containing constants for code table identifiers used in the code table XML file. This file can be used in conjunction with the `curam.util.CodeTable` interface to access code table information programmatically.

The *Java* file is generated to `/build/svr/codetable/gen/<package>` directory, where `<package>` is the package attribute specified in the codetable file. For example, *package="curam.codetable"* would result in the *Java* artefacts being placed in the `/build/svr/codetable/gen/curam/codetable` directory.

The code table SQL file contains inserts for the `CodeTableHeader` and `CodeTableItem` database tables. All SQL file artefacts are placed in a common directory: `/build/svr/codetable/sql/`.

> 🛈 Note

If code table files of the same name exist in different components with different `package` attribute values then the symbol definition file (*Java* class) artefacts are placed in the package specified by the code table file of the component with the highest precedence order (as listed in the `SERVER_COMPONENT_ORDER` environment variable).

These artefacts are best illustrated by example:

```
package curam.codetable;

/**
 * Generated AcceptanceStatus codetable file.
 *
 */
public final class ACCEPTANCESTATUS {

  /**
   * TABLENAME=AcceptanceStatus.
   */
  public static final String TABLENAME
      = new String("AcceptanceStatus");

  /**
   * DEFAULTCODE=ACS1.
   */
  public static final String DEFAULTCODE
      =  new String("ACS1");

  /**
   * Retrieves the defaultCode from the cache.
   *
   * @returns the default code value
   *
   * @throws curam.util.exception.AppException
   *    Generic Exception Signature.
   * @throws curam.util.exception.InformationalException
   *    Generic Exception Signature.
   */
  public static String getDefaultCode()
    throws curam.util.exception.AppException,
           curam.util.exception.InformationalException {
    return curam.util.type.CodeTable.getDefaultItem(TABLENAME);
  }

  /**
   * ACS1=Accepted.
   */
  public static final String ACCEPTED
      = new String("ACS1");
  /**
   * ACS2=Provisional.
   */
  public static final String PROVISIONAL
      = new String("ACS2");
  /**
   * ACS3=Rejected.
   */
  public static final String REJECTED
      = new String("ACS3");
  /**
   * ACS4=Unknown.
   */
  public static final String UNKNOWN
      = new String("ACS4");
}
```

Example 10.16 Sample Java file produced from code table file

It should be noted that this pattern of generation means that the Strings will not be interned by the *Java* compiler. This allows the dependency checking in the build scripts to operate correctly. If an empty string is provided for a *Java* Identifier the code is only mapped into persistent data (SQL file) and is not reflected in the *Java* artefacts.

The persistent data associated with code tables is generated into the common `/build/svr/codetable/sql/` directory.

```
--
-- Cúram Code Table SQL Data File
--


--
-- CODETABLE AcceptanceStatus
--
INSERT INTO CodeTableItem (TABLENAME, CODE, DESCRIPTION,
    ANNOTATION, ISENABLED, SORTORDER, LOCALEIDENTIFIER,
    LASTWRITTEN)
  VALUES ('AcceptanceStatus', 'ACS1', 'Accepted', '', '1',
    0, 'en_US', CURRENT_TIMESTAMP(''));
INSERT INTO CodeTableItem (TABLENAME, CODE, DESCRIPTION,
    ANNOTATION, ISENABLED, SORTORDER, LOCALEIDENTIFIER,
    LASTWRITTEN)
  VALUES ('AcceptanceStatus', 'ACS2', 'Provisional', '', '1',
    0, 'en_US', CURRENT_TIMESTAMP(''));
INSERT INTO CodeTableItem (TABLENAME, CODE, DESCRIPTION,
    ANNOTATION, ISENABLED, SORTORDER, LOCALEIDENTIFIER,
    LASTWRITTEN)
  VALUES ('AcceptanceStatus', 'ACS3', 'Rejected', '', '1',
    0, 'en_US', CURRENT_TIMESTAMP(''));
INSERT INTO CodeTableItem (TABLENAME, CODE, DESCRIPTION,
    ANNOTATION, ISENABLED, SORTORDER, LOCALEIDENTIFIER,
    LASTWRITTEN)
  VALUES ('AcceptanceStatus', 'ACS1', 'Passed', '', '1',
    0, 'en_GB', CURRENT_TIMESTAMP(''));
INSERT INTO CodeTableItem (TABLENAME, CODE, DESCRIPTION,
    ANNOTATION, ISENABLED, SORTORDER, LOCALEIDENTIFIER,
    LASTWRITTEN)
  VALUES ('AcceptanceStatus', 'ACS2', 'Pending', '', '1',
    0, 'en_GB', CURRENT_TIMESTAMP(''));
INSERT INTO CodeTableItem (TABLENAME, CODE, DESCRIPTION,
    ANNOTATION, ISENABLED, SORTORDER, LOCALEIDENTIFIER,
    LASTWRITTEN)
  VALUES ('AcceptanceStatus', 'ACS3', 'Failed', '', '1',
    0, 'en_GB', CURRENT_TIMESTAMP(''));
INSERT INTO CodeTableItem (TABLENAME, CODE, DESCRIPTION,
    ANNOTATION, ISENABLED, SORTORDER, LOCALEIDENTIFIER,
    LASTWRITTEN)
  VALUES ('AcceptanceStatus', 'ACS4', 'Unknown', '', '1',
    0, 'en', CURRENT_TIMESTAMP(''));

INSERT INTO CodeTableHeader (TableName, TimeEntered,
DefaultCode, LASTWRITTEN)
  VALUES ('AcceptanceStatus', CURRENT_TIMESTAMP(''),
  'ACS1', CURRENT_TIMESTAMP(''));
```

Example 10.17 Sample SQL file produced from code table file

> ℹ **Note**
>
> If any `<locale>` entries specify a language (and optional country) which are not contained in the SERVER_LOCALE_LIST environment variable they will be ignored during generation and a warning will be produced.

> Also, while generating the codetable SQL artefacts containing the contents for the CodeTableItem and CodeTableHeader database tables, the LASTWRITTEN field with an initial value will be populated. The initial value is a time stamp which is set to the time when the data is inserted into the database.

The same artefacts are produced for the code table file of Example 10.15, *Resulting Code Table File 2*, also, because the file contains a <displaynames> element, additional artefacts are created, i.e. a properties file is generated for each <name> element it contains.

The ctgen target produces one properties file for each locale (composite of language and country attributes) and <name> element within the <displaynames> element of a code table definition. Locale is defined by the language and country attributes of the <name> element. These properties files define the display names associated with each code table in a code table hierarchy.

The properties files are generated into / build/svr/codetable/gen/. If no <displaynames> element is specified for a code table hierarchy, no properties file is generated, and a warning will be displayed. The name of the generated properties file consists of the code table name along with the locale. Since a code table name with spaces renders a properties file invalid and unlocalizable, any spaces specified in the code table name will be replaced with the underscore character.

The warning, i.e. warning where a <displaynames> element is not specified, is only treated as a warning and never an error, regardless of the setting of the property prp.warningstoerrors.

If the locale specified for the <name> element is not supported, then the ctgen will display a warning and no properties file for that locale will be generated.

The following is an example of properties files produced by the ctgen on the Example 10.15, *Resulting Code Table File 2*. Each properties file is generated to /build/svr/codetable/gen/

```
CarMake=Car Make US
```

Example 10.18 CarMake_en_US.properties

```
CarMake=Marque
```

Example 10.19 CarMake_fr.properties

```
CarMake=Car Make CustomGB
```

Example 10.20 CarMake_en_GB.properties

```
CarMake=Masinos Gamintojas
```

Example 10.21 CarMake_It.properties

```
CarMake=Car Make Custom
```

Example 10.22 CarMake_en.properties

```
CarModel=Car Model
```

Example 10.23 CarModel_en.properties

10.3.4 Code Table Hierarchy

Code table files can define a single code table or a hierarchy of code tables. A hierarchy is where multiple code tables are linked into a number of levels. Selecting a code at a particular level will reduce the number of selections available at the next level. Any number of levels in a code table hierarchy is supported.

For example, Table 10.11, *Address Hierarchy* shows a sample hierarchy. Selecting Ireland as the country will return a sub-list of Meath and Wexford and selecting Meath as the county will return sub-list of Trim and Navan. Alternatively, selecting England will return a sub-list of Stafford and London, etc.

Level 1	Level 2	Level 3
Country	County	Town
Ireland	Meath	Navan
		Trim
	Wexford	Gorey
		Enniscorthy
England	Stafford	Bednall
		Stone
	London	Earlsfield
		Eltham

Table 10.11 Address Hierarchy

To define a code table hierarchy a code table (CTX) file should be created with a code table defined for each level in the hierarchy. To indicate that the code table file contains a hierarchy, the hierarchy_name attribute should be defined on the <codetables> element.

```
<codetables package="curam"
             hierarchy_name="AddressHierarchy">
<description>
  A description of the hierarchy.
</description>
```

Example 10.24 Usage of hierarchy_name attribute

Each <codetable> defined must then be linked using the `par-ent_codetable` attribute of the <codetable> element. The `par-ent_codetable` value should be set to the `name` of an existing <codetable> in the file, where the specified code table is the parent in the hierarchy. All code tables defined in the file, excluding the top level code table, must have a valid `parent_codetable` attribute defined for them. A <codetable> can be linked to only one parent <codetable> and cannot be used in more than one code table hierarchy.

```
<codetable java_identifier="COUNTY"
           name="County" parent_codetable="Country">
```

Example 10.25 Usage of parent_codetable attribute

Each <code> entry in a code table is finally linked to a <code> entry in the parent code table, using the `parent_code` attribute. The `par-ent_code` value must be the value of a <code> existing in the specified parent code table. A child <code> cannot be linked to more than one parent <codetable>.

```
<code java_identifier="MEATH"
      value="MEATH" parent_code="IRELAND" status="ENABLED">
```

Example 10.26 Usage of parent_code attribute

The hierarchy defined in Table 10.11, *Address Hierarchy* can be represented as follows in a code table file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<codetables package="curam" hierarchy_name="AddressHierarchy">
  <description>
    A description of the hierarchy.
  </description>

  <codetable java_identifier="COUNTRY" name="Country">
    <displaynames>
      <name language="en">Country</name>
    <displaynames>
    <code java_identifier="IRL" value="IRLND" default="true"
          status="ENABLED">
      <locale language="en" sort_order="1">
        <description>Ireland</description>
      </locale>
    </code>
    <code java_identifier="GB" value="ENGLND" status="ENABLED">
      <locale language="en" sort_order="2">
        <description>England</description>
      </locale>
    </code>
  </codetable>

  <codetable java_identifier="COUNTY" name="County"
             parent_codetable="Country">
    <displaynames>
      <name language="en">County</name>
    </displaynames>
    <code java_identifier="MEATH" value="MTH"
          parent_code="IRLND" status="ENABLED">
      <locale language="en" sort_order="1">
```

```
          <description>Meath</description>
        </locale>
      </code>
      <code java_identifier="WEXFORD" value="WXFD"
            parent_code="IRLND" status="ENABLED">
        <locale language="en" sort_order="1">
          <description>Wexford</description>
        </locale>
      </code>
      <code java_identifier="STAFFORD" value="STFFRD"
            parent_code="ENGLND" status="ENABLED">
        <locale language="en" sort_order="1">
          <description>Stafford</description>
        </locale>
      </code>
      <code java_identifier="LONDON" value="LNDN"
            parent_code="ENGLND" status="ENABLED">
        <locale language="en" sort_order="2">
          <description>London</description>
        </locale>
      </code>
  </codetable>

  <codetable java_identifier="TOWN" name="Town"
             parent_codetable="County">
    <code java_identifier="NAVAN" value="NVN"
          parent_code="MTH" status="ENABLED">
      <locale language="en" sort_order="2">
        <description>Navan</description>
      </locale>
    </code>
    <code java_identifier="TRIM" value="TRM"
          parent_code="MTH" status="ENABLED">
      <locale language="en" sort_order="2">
        <description>Trim</description>
      </locale>
    </code>
    <code java_identifier="GOREY" value="GRY"
          parent_code="WXFD" status="ENABLED">
      <locale language="en" sort_order="2">
        <description>Gorey</description>
      </locale>
    </code>
    <code java_identifier="ENNISCORTHY" value="ENC"
          parent_code="WXFD" status="ENABLED">
      <locale language="en" sort_order="2">
        <description>Enniscorthy</description>
      </locale>
    </code>
    <code java_identifier="ELTHAM" value="ELTM"
          parent_code="LNDN" status="ENABLED">
      <locale language="en" sort_order="2">
        <description>Eltham</description>
      </locale>
    </code>
    <code java_identifier="EARLSFIELD" value="ELFD"
          parent_code="LNDN" status="ENABLED">
      <locale language="en" sort_order="2">
        <description>Earlsfield</description>
      </locale>
    </code>
    <code java_identifier="BEDNALL" value="BDNL"
          parent_code="STFFRD" status="ENABLED">
      <locale language="en" sort_order="4">
        <description>Bednall</description>
      </locale>
    </code>
    <code java_identifier="STONE" value="STN"
          parent_code="STFFRD" status="ENABLED">
      <locale language="en" sort_order="4">
        <description>Stone</description>
      </locale>
```

```
    </code>
  </codetable>
</codetables>
```

Example 10.27 Code Table Hierarchy Example

The artefacts listed in Section 10.3.3, *Artefacts Produced by ctgen Build Target* are also generated for code table files that define a hierarchy.

Properties files are generated for `<displaynames>` elements. A symbol definition *Java* file is generated for each code table in the hierarchy. A single SQL file is generated, containing the relevant inserts to the `CodeTableHeader` and `CodeTableItem` database tables for all defined code tables. These insert statements will include the population of the `parentCode` field in the `CodeTableItem` table and the `parent_codetable` field in the `CodeTableHeader` table. An insert entry is also generated for the `CodeTableHierarchy` database table. This table is used for administration purposes only.

> ℹ️ **Note**
>
> The code table hierarchies can only be created through code table (CTX) files and not through the admin screens. The admin screens can only be used to maintain the code table hierarchies.

10.3.5 Retrieving Codes from Code Table Files

Since a code table file can contain any number of locales for a named code a mechanism needs to be in place to return the correctly localized code for a running instance of Cúram. Codes are retrieved from a code table file based on the locale property which includes a language and or a country. The code table file look up returns a matching localized code for a named value. For example, if the runtime locale is set to `en_US` where "en" is the language and "US" is the country, a code look up for the code named ACODE in Example 10.28, *Code File Search*, will return the text "The code". If, however, the runtime locale was set to "fr", the text "Le code" would be returned.

```
<codetables package="curam.codetable">
  <codetable java_identifier="AN_ID" name="ANAME">
    <code default="true" java_identifier="ACODE"
          status="ENABLED" value="ACODE">
      <locale language="en" country="US" sort_order="0">
        <description>The code</description>
        <annotation></annotation>
      </locale>
      <locale language="fr">
        <description>Le code</description>
        <annotation></annotation>
      </locale>
    </code>
  </codetable
</codetables>
```

Example 10.28 Code File Search

Since code table files are not guaranteed to contain an entry for every coun-

try, a fall back mechanism is in place. Once a code of a given name has been found and there is no direct match with the specified locale, the rules for fall back are as follows:

- If the runtime locale is set to include a language and country, the country is removed and the search looks for a matching language only. Looking up the code named ACODE in Example 10.28, *Code File Search*, with runtime locale `fr_CN` would return the text "Le code".

There is no default behavior for code tables. If the runtime locale does not find a match in the code table after applying the fall back rule an empty string is returned.

10.3.6 Localizing SDEJ Code Table Files

It is possible to localize or modify the codetable files shipped with the SDEJ. These codetable files are located in the `codetable` directory of the SDEJ and are in the same format as Cúram application codetable files but with the extension `.itx`.

To localize these files copy the particular `.itx` codetable file to be modified from the SDEJ to the codetable directory of a component in your Cúram application, for example, `SERV-ER_DIR/components/custom/codetable`. The `.itx` codetable file can then be modified in the same way as any codetable file; overriding a code or adding a new locale for all the codes.

> ℹ **Note**
>
> If the `package` attribute in the codetable file is modified the localization will not work.

The `ctgen` target, when run, will merge the localized `.itx` codetable file with the original one located in the SDEJ. The localized codetable file will have the higher precedence order. It will then generate the sql files only. No *Java* artefacts will be generated for codetable files with the extension `.itx`.

The `datamanager_config.xml` file, located in the `project/config` directory specifies the location of the common directory for generated SQL artefacts. There is no requirement to update this entry for localized code tables as all `.sql` files are generated to the same location.

```
< entry
  name="build/svr/codetable/sql/"
  type="sql"
  base="basedir"/>
```

Example 10.29 Datamanager entry for the code table SQL artefacts location

> ℹ **Note**
>
> The <description> sub-element is an optional element for the

<codetables> element in the codetable (CTX) files. The <description> element is mainly used to define a description for the code tables for developers information. The description is not saved into any database tables for normal code tables. However, for Code Table Hierarchies, if the description is defined in the CTX file, then the <description> value is saved to the description attribute in the CODETABLEHIERARCHY table. This value will be displayed on the Code Table Hierarchy page of the Cúram Administration screens.

Notes

[1]See Section 10.2.2, *Customizing a Message File*, for further explanation of `SERVER_COMPONENT_ORDER`.

Chapter 11

Specialized Readmulti Operations

11.1 Overview

Generated readmulti operations in *IBM Cúram Social Program Management* servers execute SQL SELECT statements and return the resulting record set as an ArrayList. In fact, readmulti operations are implemented as two very distinct pieces:

- a Data Access Layer function which establishes the result set, through building up the statement, executing an `executeQuery` on it, and finally a series of `getResultObject` statements, and

- a Business Object Layer function which assembles the results into the required in-memory vector of structures.

The Business Object Layer function is a specialization of a general class of functions called readmulti operations, which can perform arbitrary processing on the contents of SQL cursors. You can view the definitions of these function classes in `curam.util.dataaccess.ReadMultiOperation`. This `ReadMultiOperation` is the parent abstract class, while `curam.util.dataaccess.StandardReadMultiOperation` is a concrete subclass providing an implementation of "normal" readmulti functions.

"Specialized readmulti operations" are simply hand-crafted functions "plugged into" the Data Access Layer using generated helper classes. The pattern in use here is similar to the *Visitor* design pattern described in *Design Patterns* by Gamma et al. Readmulti operations are "plugged into" the appropriate Data Access Layer functions by generated readmulti helper classes, which insulate the operation from knowledge about the specific Data Access Layer functions used.

11.2 When to Use Readmulti Operations

"Normal" readmulti operations return a set of database records as an `ArrayList`. There are several situations in which you might want to replace this type of standard "normal" readmulti operation with your own specialized processing.

An example is in batch processing where you want to iterate across a large number of records on a database table, and process each record in turn. It is not feasible to use a standard readmulti operation to assemble an in-memory vector of all of the records read before processing. Another common example is where you want to lock or delete records from the result set as they are processed. In each of these examples you can write your own readmulti operations to process records individually as they are retrieved from the database rather than relying on the standard processing supplied by `StandardReadMultiOperation`.

11.3 How to Define Your Own Readmulti Operations

The steps that you follow to define your own specialized readmulti operations are as follows:

1. Add the readmulti operation to your UML application model. We will assume for this example that you add a standard readmulti operation called `readmulti` to an entity called `E`. The standard readmulti operation whose "details" structure will be called `EDtls`. However, this example applies equally to `<<readmulti>>`, `<<nsreadmulti>>`, `<<nkreadmulti>>` and `<<nsmulti>>` operations in the UML application model, where the "details" structure might not be a generated entity details structure.

2. Write the specialized readmulti operation class, as follows:

```
static class MyReadmultiOperation extends
curam.util.dataaccess.ReadmultiOperation {

  public boolean operation(Object objDtls) throws
    AppException, InformationalException {

  // No implementation for the moment

  return true;

  }

}
```

> **Note**
>
> If the readmulti operation specifies a 'Post Data Access' or 'On-fail' operation then your readmulti operation must be a subclass of
> `curam.util.dataaccess.StandardReadMultiOpe`

ration. This is because this class builds up an in-memory list of the structs which are read by the readmulti operation in order to make it available to the `Post Data Access` and `On-fail` operations.

If your readmulti operation processes large numbers of records then this could cause an excessive memory usage overhead; so caution is advised if using specialized readmulti operations in conjunction with `Post Data Access` or `On-fail` operations.

3. Implement `MyReadMultiOperation.operation` to perform your specific processing. This method will be called automatically for each record retrieved from the database.

    In general, always return `true` from `readmulti` operations. In unusual cases, where you want to stop processing before you hit the end of the record set, return `false`. This means the operation method will not be called again.

4. Write the code that will invoke the readmulti operation. This will appear in a BPO implementation and look like this:

```
// instance of specialized operation class
MyReadMultiOperation op = new MyReadMultiOperation();

// instance of readmulti key structure
EReadmultiKey key;

// set key fields for search
key.id = 99;

// construct helper and call operation
E.newInstance().readmultiHelper(key, op);
```

Each generated readmulti function is associated with a generated "helper" class which exists solely for use in code like that above. The helper class is scoped inside the entity class and has an execute method that begins a readmulti.

11.4 Extra Features of Readmulti Operations

*   The `READMULTI_MAX` option in the model limits the number of records processed by a standard "normal" readmulti operation. It has however no effect when you hand-craft your own operations. As a result none of the overrides for this option (defined in Chapter 4, *Cúram Configuration Settings*) have any effect. To limit the number of records returned within your readmulti subclass you must override the following method:

    `public int getMaximum();`

*   You can filter out records from the database result set by overriding the following method of your readmulti subclass:

```
public boolean filter(Object dtls) throws AppEx-
ception, InformationalException;
```

Each record is passed to `filter` before being passed to your `opera-tion` method. Any record which results in `filter` returning `false` is not passed to `operation`. The default `filter` always returns `true`.

- If you want to write code that is called before the first row is passed to `operation`, you can override:

```
public void pre() throws AppException, Informa-
tionalException;
```

If you want to write code that is called with the first row read from the database, you can override:

```
public void first(Object dtls) throws AppExcep-
tion, InformationalException;
```

The same record is also passed to the `operation` method.

> **Note**
>
> Note that `first` is called as long as there is at least one row in the result set, regardless of whether or not `filter` returns `true` for this row.

- If you want to write code that is called after the last call to `operation`, you can override:

```
public void post() throws AppException, Informa-
tionalException;
```

Be aware, this function is always called once, regardless of the value returned by the `operation` method.

- An optional third parameter to the `execute` method of readmulti helper classes is a `boolean` which specifies whether records read from the database will be updated later in the transaction. This can be used as in :

```
E.newInstance().readmultiHelper(key, op, true);
```

This means that each record read from the database is locked for write access as it is read.

You can use a combination of the above methods, with your own data members, to achieve many common styles readmulti operation. For instance, Example 11.1, *Specialized readmulti example* shows a readmulti operation that produces a report grouped by department:

```
static class MyReadmultiOperation
        extends curam.util.dataaccess.ReadmultiOperation
{
  // Remember last dept, for grouping
  private String lastDepartment;

  // Department salary accumulator
```

```
  private curam.util.type.Money salaryDeptTotal;

  // Total Salary Accumulator
  private curam.util.type.Money salaryGrandTotal;

  public void pre()
    throws AppException, InformationalException {
    // initialization
    lastDepartment = "";
    salaryGrandTotal = 0.0;
  }

  public void first (Object dtls)
  throws AppException, InformationalException {
    // per-department group initialization
    salaryDeptTotal = 0.0;

    // remember last department for grouping.
    lastDepartment = dtls.department;
  }

  public boolean operation(Object dtls)
    throws AppException, InformationalException {

    // Change of department group
    if (!(lastDepartment.equals(dtls.department))) {
      printGroupTotals();

      // redo per-dept initialization
      first(dtls);
    }

    // detail report line
    curam.util.resources.Trace.kTopLevelLogger.info("Emp ");
    curam.util.resources.Trace.kTopLevelLogger.info(
      dtls.employeeId);
     curam.util.resources.Trace.kTopLevelLogger.info(
      " salary: ");
    curam.util.resources.Trace.kTopLevelLogger.info(
      dtls.salary);

    // accumulate dept salary
    salaryDeptTotal += dtls.salary;

    // accumulate total salary
    salaryGrandTotal += dtls.salary;

    return true;
  }

 public void post()
    throws AppException, InformationalException {
    // only if there was at least one department
    if (!(lastDepartment.empty())) {
      printGroupTotals();
      // final group
      // Grand total report line:
      curam.util.resources.Trace.kTopLevelLogger.info(
        "Grand total salary: ");
      curam.util.resources.Trace.kTopLevelLogger.info(
        salaryGrandTotal);
    }
  }

  public int getMaximum() {
    // Explicitly enforce that all matching records are
    // considered.  Any number other than zero would limit
    // the number of records.
    return 0;
  }

  private void printGroupTotals() {
```

```
    // group report line
    curam.util.resources.Trace.kTopLevelLogger.info(
      "Department ");
    curam.util.resources.Trace.kTopLevelLogger.info(
      lastDepartment);
    curam.util.resources.Trace.kTopLevelLogger.info(
      " total salary: ");
    curam.util.resources.Trace.kTopLevelLogger.info(
      salaryDeptTotal);
  }
}
```

Example 11.1 Specialized readmulti example

11.5 An Alternative

Specialized Readmulti operations and non-standard operations allow the developer a greater level of freedom when handcrafting database access code. However in certain situations this may prove to be too limiting. For example where the SQL string will be derived from the input parameters to a method; parts of the 'where' clause will be optional or expressed differently depending on the input. In these situations the developer can obtain the Connection being used for database communication through the `TransactionInfo.getInfoConnection` interface. Once this connection has been obtained it is possible to execute any form of handcrafted JDBC in the context of the Cúram transaction.

To enable this style of database access to be visible in the model it should be placed in an entity which has the `NO_SQL` option enabled. This is detailed in the *Cúram Modeling Reference Guide*.

11.6 Summary

The order in which your readmulti operation methods are called is:

- `pre`- always called once before anything else;

- `first`- called once with the first record, provided at least one record exists;

- `filter`- called for each record (including the first);

- `operation`- called for each record for which filter returns `true`;

- `post`- always called once after everything else;

- `getMaximum`- specifies the maximum number of records that should be matched.

If you are designing processing that maintains locks remember that there are performance implications when you do so.

Chapter 12

Deprecation

12.1 Introduction

IBM®IBM Cúram Social Program Management uses deprecation as a means of reducing the impact of change on custom applications. This chapter describes deprecation in Cúram: what it is, how it can affect custom code, what it means for support and the build infrastructure that helps pinpoint custom artefact dependencies on deprecated Cúram artefacts.

12.2 Overview

In enhancing Cúram in a Major Version release or fixing defects in a Service Pack, the necessity occasionally arises where the contract of a Cúram development artifact has to be changed. In this context the contract of an artifact is its API or signature (e.g. name, parameters, return values, etc) in conjunction with its documented statement of functionality (e.g. JavaDoc).

Prior to Cúram 6.0, such changes would typically have been made in place, potentially causing compilation errors or unexpected runtime behavior in a custom application. This policy changed in 6.0 to favor adding a new artifact which implements the changed behavior while preserving the original artefact and marking it 'deprecated'. This has two main benefits for custom applications. Primarily, it provides back-compatibility for any references in custom code to the deprecated artefact (n.b. it does not provide back-compatibility for a custom override of the deprecated artefact). It also eliminates a source of compilation failures during upgrades, which can hamper the development of a reliable upgrade plan. These effects are described in more detail in this chapter.

Finally, infrastructure is provided in Cúram that extends *Java*'s command-line compiler deprecation warnings to certain Cúram builds. This helps pinpoint dependencies in custom applications on deprecated Cúram artefacts. It also helps distinguish between references-to and customizations-of deprec-

ated artefacts in custom code. That build infrastructure is also described in this chapter.

12.2.1 Other Sources of Information

Information about specific deprecated artefacts, can be found in the artefact itself and also in the *'Notes on Deprecation'* section of the Cúram release notes.

In the artefact itself, the deprecated element will be marked as described in Section 12.4.1, *Artefact Types that can be Deprecated*. This marker includes space for a short 'deprecation comment' about the replacement functionality for the deprecated item and a reference to any associated release note containing more context. To make your analysis easier, Cúram validation and compilation steps will include this comment in the build warning, to save you looking up the deprecated artefact. However, this enhanced build warning is only available from Cúram compilers/validations, the command-line *Java* compiler does not have equivalent functionality. It is recommended you view *Java* warnings in your IDE for fast navigation between artefacts.

If the information in the artefact's deprecation comment does not provide enough context, additional information can be found in the Cúram Release Notes. You can search these by the name of the deprecated artefact or by the release note ID referenced in its deprecation comment.

12.3 Effect of Deprecation on a Custom Application

In Cúram, a 'deprecated' artefact means an artefact that has been replaced by other functionality and is no longer part of the default flow of Cúram. Deprecated artefacts remain present in the application codebase, but they are not referenced by the out-of-the-box runtime application. If deprecated artefacts are referenced in the out-of-the-box application codebase, it is only by other deprecated artefacts.

To quickly pinpoint where custom dependencies exist on deprecated Cúram artefacts, the command-line *Java* compiler has been extended to provide deprecation warnings to Cúram builds and validations. This will be described in more detail later in this chapter.

12.3.1 Customizations and References

Custom artefacts can depend on deprecated Cúram artefacts either by referencing them, or by customizing (overriding) them. Reference and customization dependencies have different characteristics and it is important to understand the difference. To illustrate:

- Examples of References

  - A custom method can call a deprecated Cúram server interface method

- A custom workflow can reference a deprecated Cúram method as an automatic activity

- A custom UIM client page can link to a deprecated Cúram UIM page

- Examples of Customizations

  - A custom class can subclass a Cúram class and replace (override) deprecated Cúram methods

  - A custom UIM client page can customize (override) a deprecated Cúram UIM client page

The impact of deprecation on custom code depends on whether that code is referencing or customizing a deprecated artefact.

Where code *references* a deprecated Cúram artefact (e.g. calls a deprecated method), the deprecated artefact still exists and functions in a backwardly-compatible way. This is the same as for regular *Java* deprecation where the immediate impact is minimal or nil.

Where code *customizes* (overrides) a deprecated Cúram artefact, the base Cúram Application no longer invokes that artefact - it is no longer part of the "default flow" of the base application. It is reasonably likely that it has been removed from the default flow of custom applications. In short, customizations of deprecated artefacts do not function as before and there is a strong likelihood that some corrective action will be needed. That action could include dropping the customization (e.g. if equivalent functionality has since been implemented), re-applying the customization to the artefact that replaces the deprecated one, etc.

The deprecation build infrastructure provided uses special tags in deprecation warnings to help distinguish between references-to and customizations-of deprecated artefacts. This will be described in more detail later in this chapter.

12.3.2 Support for Deprecated Artefacts

Deprecated artefacts will continue to be supported as long as the version in which they were deprecated remains in-support. Customer-raised defects will continue to be addressed in the deprecated artefacts during this period. All future enhancement requests will be directed toward the replacement functionality.

Deprecation of an artefact is an indication of the intent to remove it in a future version. However, a deprecated artefact will remain in- support as long as the version it was deprecated in remains in-support. After that, it is subject to removal without further notice. Extended support for specific artefacts will be considered as long as the request is made in good time (no less than 6 months in advance of the relevant version falling out of general support).

You are advised to address any dependencies from Custom code on deprecated Cúram artefacts at the earliest opportunity. The replacement functionality offers better support and better upgrades in future. When deprecated artefacts are removed in a future release, it can cause compilation failures and this can seriously hamper effective planning of upgrade tasks.

12.3.3 Effect of Deprecation on the User Interface

When client pages are deprecated, this changes the default flow of the client application to include the replacement functionality. This has two results that do not occur when other artefacts are deprecated:

Consistency of the User Interface: If existing client pages have been customized or new pages added which are used in conjunction with deprecated pages, then the resultant user experience may be changed with the replacement pages. If this is the case it will be necessary to consider updating the customizations to be consistent with the replacement pages, or reverting the default flow to use the deprecated pages.

Documentation/Training Materials: If descriptions and/or screen shots of the deprecated pages have been included in custom documentation/training materials, these may need to be updated to describe/show the replacement pages.

12.4 Scope

12.4.1 Artefact Types that can be Deprecated

The following artefact types may be deprecated:

Area	Artefact Type
Modeled Artefacts	Process Class, Entity Class, Struct Class, Process Method, Entity Method.
Java Code	Identical to *Java* deprecation (Class, Interface, Method, Attribute, etc)
Client Artefacts	UIM Page, VIM file, Page Property (.property associated with a UIM or VIM file)
Messages	Message Catalog Entry

Table 12.1 Artefact Types that can be Deprecated

All of these artefact types support explanatory comments attached to the deprecation tag. These can be easily found by searching for the string 'deprecated' within the artefact in question. For `.java` files (and model artefacts), the **@deprecated** JavaDoc tag is used in the normal way. For XML files such as UIM/VIM files and message catalog entries, the **<?curam-deprecated** XML processing instruction is used. Finally, in prop-

erty files, the string **.deprecated** is appended to the name of a property to denote that that property is deprecated.

> ℹ **Entity Classes**
>
> Please note that for Entity Classes, the term 'deprecation' refers entirely to the generated *Java* artefacts derived from the Entity and does not refer to data associated with that entity. The Cúram Generator produces no database schema representation for a deprecated Entity, it is expected that this data has migrated to another Entity (or Entities). Entities are deprecated in order to minimize the *code* impact of changes to the data model. The deprecated Entity's purpose is to re-route method calls to the appropriate replacement Entity (or Entities). As such, deprecated Entities follow the same pattern as other classes - references to the entity will continue to function as before, customizations (overrides) will not.

12.4.2 Limitations

There are certain limitations of the deprecation infrastructure to be aware of:

- No build warnings will be produced for non-typed references to deprecated artefacts. For example, if the UIM page `Participant_viewAddress.uim` was deprecated and a *Java* method contained a "Participant_viewAddress" string literal - this would not be picked up by the build warnings because the reference is not typed - the compiler cannot know that the String refers to a UIM page.

- The deprecation infrastructure is comprised of a deprecation tagging capability and build/validation warning capability (reporting dependencies on tagged artefacts). The build/validation warning capability is intended for customer use. The deprecation tagging capability is not intended for customer use and is therefore not supported. For example, using the **<?curam-deprecated** processing instruction in custom XML files is not supported.

12.5 Running a Deprecation Report

Cúram has developed infrastructure that extends *Java*'s command-line compiler deprecation warnings to certain Cúram builds. This helps pinpoint dependencies in custom applications on deprecated Cúram artefacts. It also helps distinguish between references-to and customizations-of deprecated artefacts in custom code.

12.5.1 Configuring the Deprecation Report

Deprecation reporting in Cúram is controlled by two properties:

- Ensure the **prp.warningstoerrors** build property, is set to *false* or the build may be unable to complete (*false* is the default for this property, so

if you do not override the property then the default is fine).

- The **curam.deprecation.reporting** property in the `boot-strap.properties` file controls the reporting of deprecation warnings. Warnings are not displayed if this property is set to *false*. The property defaults to *true* so if it is not specified deprecation warnings will be displayed.

- It is recommended you remove "Sample" components (Sample, CPMSample, etc) from the CLIENT_COMPONENT_ORDER environment variable before running the commands below. These components may generate spurious warnings that are not relevant to identifying your exposure to deprecated Curam artefacts.

12.5.2 Prerequisites for running the Deprecation Report

The deprecationreport build target calls a sequence of Cúram build targets in order to provide build output containing a complete set of deprecation warnings. As there are dependencies between some of the build steps the following builds should complete successfully before running the deprecationreport target.

- build clean server

- build clean client

- build database

12.5.3 Generating the Deprecation build output

Execute the build target below, it will capture the build output to a `%SERVER_DIR\buildlogs\%Deprecation<timestamp>.log` file for further analysis.

- **cd %SERVER_DIR%**

- **build deprecationreport**

12.5.4 Identifying deprecation warnings in the build output.

Since the build output has all been directed into the `Deprecation<timestamp>.log` file, check that file, to ensure that the overall build succeeded. *Ant* prints either a 'BUILD SUCCESSFUL' marker in the last few lines of that file if all parts of the build completed (or 'BUILD FAILED' if any failed).

Since you have already confirmed that the server, client and database builds complete successfully, the only issues that are expected to cause this target to fail are validation issues. Since the validation of one file has no bearing on the next, the these targets do not stop on a failed validation. They aim to

provide as complete a picture as possible by validating all files and only re-porting success or failure at the end of the build. So the deprecation inform-ation will still be produced for all files that pass validation.

Finally, to get a summary report of all exposure to deprecated artefacts, fil-ter the `deprecation.log` for the `[deprecation]` tag. You can use **grep** or the *Windows* **find** utility for this, or your preferred text editor. e.g.:

```
grep "\[deprecation\]" Deprecation<timestamp>.log
1> deprecation_summary.log 2>&1
or
find "[deprecation]" Deprecation<timestamp>.log
1> deprecation_summary.log 2>&1
```

Example 12.1 Getting a Summary Report

The resulting `deprecation_summary.log` file will contain only the deprecation warnings produced by the build.

> **Tip:**
>
> Since some warnings can be broken over more than one line, it is useful to hold on to the original `deprecation.log` as well.

12.5.5 Notes on running the Deprecation Report

- This build can take some time to run, as it has to do a clean followed by server and client build, in order to identify all dependencies. The target also does the validations for several artifact types.

- Although the **deprecationreport** target generates the `deprecation build log`, it is not always necessary to rerun the entire build in case it fails. If the build fails due any validation, the validation target can be ran in isolation. After fixing all the validation issue, **deprecationreport** target should be executed to ensure the deprecation build log is com-plete.

- The **deprecationreport** calls the **validation** target. For example: The **deprecationreport** will fail if **validateallworkflows** target will report an error, as the build output from other builds is not available.

```
[deprecation] The client has not been built and therefore it
cannot be determined if UIM pages referenced are
deprecated.
```

- By default the *Java* compiler limits the number of compiler warnings displayed. The Cúram build specifies this limit as 10,000, which means that the compiler will display 10,000 warnings followed by a message that, there were further warnings. This value can be overridden by passing **-Dcmp.maxwarnings** to the build.

- IEG scripts can also contain dependencies on server and/or client arti-

facts that have become deprecated. However, this scenario is not covered by **validation** targets at this time. If you have IEG scripts, you will need to manually inspect UIM page and server interface references to identify any dependencies on deprecation pages or interfaces.

> ℹ️ Note
>
> Since some warnings can be broken over more than one line, it is useful to hold on to the original `deprecation.log` as well.

12.6 Analyzing Deprecation Warnings

Once you have produced a summary deprecation build log you need to identify the deprecation warnings contained in it. This section describes how to identify and categorize the deprecation warnings

12.6.1 Identifying overrides of deprecated artefacts

As described in Section 12.3.1, *Customizations and References* there are significant differences between the effects of deprecation on references and on customizations. Identifying overrides of deprecated artefacts is relatively simple. The deprecation summary report you produced in Section 12.5, *Running a Deprecation Report* pinpoints all dependencies on deprecated artefacts using the standard *Java* `[deprecation]` tag in the build log. Curam code generators and command-line validations also check for dependencies on deprecated artefacts and produce the same build warning as *Java* (using the same `[deprecation]` tag).

In addition to this, Curam code generators augment the `[deprecation]` tag with an additional `[customization]` tag where your custom artefact is overriding a Curam artefact, rather than referencing it.

Any lines in your deprecation summary report tagged with `[deprecation]` `[customization]` indicate places where you are overriding an artefact that Curam has since deprecated (i.e. removed from the default flow of the base application). Since Cúram has removed this artefact from the default flow of the out-of-the-box application, it is reasonably likely that it has also been removed from the flow of your custom application. Where this happens, it will be necessary to address the override.

The example below shows a custom VIM file that is overriding an out-of-the-box Curam VIM file. The Curam VIM file has become deprecated, so the client build is producing this warning. The warning follows the *Java* deprecation message format: the first part is the path of the file that references the deprecated artefact, followed by the [deprecation] tag and, in this case, a [customization] tag also. This is followed by the name of the artefact that has been deprecated. Finally (and this differs from the *Java* format) where possible, any comments attached to the deprecated artefact are also printed. This saves you having to locate the file and look up the associated comments.

```
[processUim]
C:/webclient/components/custom/Case_listView.vim warning:
[deprecation] [customization]
C:/webclient/components/core/Case_listView.vim has been
deprecated. [deprecation comment] Since Curam 6.0,
replaced with Case_listAnotherView.vim. See release note:
CR12345
```

Example 12.2 Example: override of a deprecated artefact

In the above example, the VIM file is no longer used in the default flow of the out-of-the-box Curam application. If your application relies on the out-of-the-box flow, your customization of this file will no longer appear in that flow.

Addressing overrides of deprecated artefacts

There is no single approach to addressing overrides of deprecated artefacts. You must analyze the modifications you made to the original (now-deprecated) artefact and determine a suitable course of action for your customization. Some options are to drop the customization (e.g. if Cúram have since implemented equivalent functionality), to re-apply the customization to the artefact that replaces the deprecated one, etc. There are sources of information that can help you when deciding the appropriate course for your customization, please see Section 12.2.1, *Other Sources of Information*

12.6.2 Identifying references to deprecated artefacts

References (e.g. calls to) to deprecated artefacts can also be easily identified in your deprecation log - they are lines tagged with a [deprecation] marker, but no [customization] marker.

```
[processUim] C:\Curam\webclient\components\custom\
Custom Benefit\Deduction\listThirdPartyDeduction.uim
warning: [deprecation] UIM ProductDelivery_cancelDeduction
has been deprecated. [deprecation comment]
Since Curam 6.0, replaced with ProductDelivery_cancelDeduction1
```

Example 12.3 Example: reference to a deprecated artefact

In the above example, the UIM page is no longer used in the default flow of the out-of-the-box Curam application and is deprecated.

12.6.3 Notes on analyzing deprecation warnings

- You should not see any deprecation warnings from out-of-the-box Curam files. However, there are instances where a deprecation issue in your custom file can appear, as if it came from an out-of-the-box Curam file. If you overrode a .VIM client file that is being used by an out-of-the-box .UIM page, any warnings from your VIM file will appear as

if they came from the out-of-the-box UIM page. This is because the client build imports .VIM content into UIM pages before validating it. If you see deprecation warnings from out-of-the-box UIM pages, please be aware that they may be referring to issues in a custom VIM file.

- If you have included sample components in your build (such as Sample, CPMSample, etc), you may also see deprecation warnings from these components. Curam does not recommend including sample components in your builds.

- You will find [deprecation comment] marker, if the tag @depreceted in documentation field has a comment. This save you having to look up the file and then look up the file it's referencing and then get the comment.

- Please be aware that any deprecation warnings marked [bopigen] in the build log are duplicates of warnings produced earlier in the log and marked as [servercodegenerator]. You can safely ignore deprecation warnings marked as [bopigen].

- Warnings coming from generated java classes (those in build/svr/gen) are by-products of the [customization] warnings produced by the generator and can generally be ignored. Resolving the "[deprecation] [customization]" issues should also resolve these generated file warnings.

> **Note**
>
> It is easier to work with java deprecation warnings in Eclipse, than it is to use the command-line deprecation build logs.

Chapter 13

User Preferences

13.1 Overview

User preferences are name-value options which specify settings that can be customized for a particular user. A set of `DefaultPreferences` is assigned to each user of the Cúram application.

A user preferences editor is available in the web client. This editor allows each user to update values for the preferences. Examples of user preference usage include setting the time zone, or providing a flag to turn a custom option on or off.

A set of user preferences are defined out-of-the-box in Cúram:

Name	Description	Default Value
Time Zone	The user's time zone.	Europe/ Dublin
High Contrast Mode	The high contrast accessibility feature adjusts the colors and images used in the application to ensure all visual content is accessible to users with limited color vision.	false

Table 13.1 Out of the box user preferences

13.2 User Preferences Definition

13.2.1 Data definition XML file

It is possible to create new user preferences, or override existing user preferences, by creating a custom `DefaultPreferences.xml` file.

165

A custom `DefaultPreferences.xml` file should be placed in the `EJBServer\components\<component_name>\userpreferences` directory, where `<component_name>` is the name of a component within the component directory.

The following sample `DefaultPreferences.xml` file illustrates how a user preference is defined:

```
<Preferences>
  <PreferenceSet id="default"
       description="The default preferences">
    <Preference name="sample.pref" category="DefaultPreferences">
      <type>SVR_BOOLEAN</type>
      <value>false</value>
      <readonly>false</readonly>
      <visible>true</visible>
      <externalVisible>false</externalVisible>
    </Preference>
  </PreferenceSet>
</Preferences>
```

Example 13.1 Example of user preference definition

In the user preferences definition example above the preference "sample.pref" is defined in an XML document with a root `Preferences` node.

The `Preferences` document may contain only one `<PreferenceSet>` element, with the `id` attribute set to "default". The `<PreferenceSet>` contains any number of `<Preference>` elements, each defining a new preference or overriding an existing one.

The `name` attribute of `<Preference>` defines the internal name of the user preference. This attribute forms a unique name for the preference stored in the database. In the example above the name is "sample.pref".

A `<Preference>` element contains a number of child elements, listed in the table below.

Element	Description	Mandatory	Default Value
type	Indicates the preference type, which should be a valid `Domain Definition` type.	yes	N/A
value	The initial default value of the user preference.	yes	N/A
readonly	A boolean value (true or false) that indicates whether the preference should be editable in the user preference editor in the web client.	no	false
visible	A boolean value (true or false) that indicates whether the preference	no	true

Element	Description	Mandatory	Default Value
	should be displayed in the user preference editor in the web client for an internal user, i.e. a user on the Users table.		
externalVisible	A boolean value (true or false) that indicates whether the preference should be displayed in the user preference editor in the web client for an external user.	no	false

Table 13.2 User Preference options

If multiple `DefaultPreferences.xml` files exist (in different components), the contents of these files are merged together during a server build. The files are merged according to the `SERVER_COMPONENT_ORDER`. Duplicated preferences in a component with higher precedence in the `SERVER_COMPONENT_ORDER` will take priority over those duplicates in components with lower precedence.

The results of the merged user preferences are added to the database by the `database` build target for usage at runtime.

> ### Note
>
> Only the default value of the out of the box user preferences in Cúram should be overridden.
>
> Although the ability to override all elements of a user preference exists it is strongly recommended that only the actual value, as defined by the `<value>some_value</value>` element, should be updated.

13.2.2 Properties files

When defining a user preference in the `DefaultPreferences.xml` file a corresponding entry should also be made in an accompanying `Default-Preferences_<locale>.properties` file. where, `<locale>` represents the intended locale of the properties. This file specifies the display name that will be displayed when accessing the user preferences in the web client user preferences editor. The ability to localize the display name for each of the user preferences is possible by creating a DefaultPreferences_<locale>.properties file for each supported user `locale`. See Section 13.5, *Localizing Display Names* for more details on localizing user preferences display names.

A `DefaultPreferences_<locale>.properties` file should be created if it does not already exist. The `DefaultPreferences_<locale>.properties` should be placed in the `EJBServ-`

er\components\<component_name>\userpreferences direct-
ory with the corresponding `DefaultPreferences.xml`.An entry for
the user preference defined in the previous example would be:

`sample.pref=Sample Preference Display Name:`

`DefaultPreferences_<locale>.properties` files in multiple
components will be merged using the same SERV-
ER_COMPONENT_ORDER merge rules that apply to `DefaultPrefer-`
`ences.xml` files.

13.3 Development Support

User Preferences can be accessed at development time using the `get-`
`Value()` and `setValue()` methods in the
`curam.util.userpreference.impl.UserPreference` class.

A user preference must have been previously created before invoking the
`setValue()` method. See Section 13.2, *User Preferences Definition* for
more details on creating user preferences.

13.4 External Users

To make user preferences available to an external user, you need to make
both client and server changes. These changes are described below.

For the client, you need to set the `USER_PREFS_PAGE` attribute to true
within a `<link>` element. Please see the *Cúram Web Client Reference
Manual* for further details on setting this element.

The `ExternalAccessSecurity` interface is used to retrieve informa-
tion for an external user. This class contains 2 new methods, `getUser-`
`PreferenceSetID()` that reads user preferences for an external user
and `modifyUserPreferenceSetID()` that updates user preferences
for an external user. These methods must be implemented in order to re-
trieve user preferences for an external user. Please see the Customizing Ex-
ternal User Applications chapter in the *Cúram Security Handbook* guide for
further details on the `ExternalAccessSecurity` interface.

Once the client and server changes have been implemented, you must en-
sure that the relevant user preferences are visible to the external user. The
`<externalVisible>` element within the `DefaultPrefer-`
`ences.xml` allows you to manage the visibility of each user preference to
an external user. This element is described in Section 13.2, *User Prefer-
ences Definition*.

If you want to make user preferences available for external users and
`<externalVisible>` is set to false or is not defined for all user prefer-
ences, then there will be no user preferences displayed for an external user.
If you do not wish to have any user preferences displayed for external users,
it is recommended that the User Preferences link should not be available
within the external user application.

13.5 Localizing Display Names

Localized display names can be added by creating new `DefaultPreferences_<locale>.properties` files for each `DefaultPreferences.xml` file created under directory `EJBServer\components\<component_name>\userpreferences`. `<locale>` represents the intended locale of the properties file and `<component_name>` is the name of a component within the component directory.

For example, to support the `en_US` locale, you should create the following default preference properties file

```
DefaultPreferences_en_US.properties
```

As there may exist multiple `DefaultPreferences_<locale>.properties` files in different components, the contents of these default preference properties will be merged to a `MergedDefaultPreferences_<locale>.properties` file according to the `SERVER_COMPONENT_ORDER` [1] . This merging happens when running either of the following targets: `mergeuserpreferenceproperties`, `server`.

Before merging the .properties files, the following validations will cause an error during a build where:

* The specified `<locale>` is not present in the `SERVER_LOCALE_LIST` [2].

* More than one display name is specified for the same locale.

  For example, two display names are specified for locale en_US.

  ```
  DefaultPreferences_en_US.properties:
    Timezone=TimeZone:
    Timezone=TimeZone US:
  ```

* The `<locale>` in the property file name includes a country part with more than 2 characters.

  For Example:

  ```
  DefaultPreferences_en_USA.properties
  ```

* The `<locale>` in the property file name includes a language part with more than 2 characters.

  For Example:

  ```
  DefaultPreferences_eng_US.properties
  ```

* The .properties file is empty.

- The .properties file contains invalid properties.

    For Example:

```
DefaultPreferences_en_US.properties:
    Timezone
```

The infrastructure will attempt to display the correct localized name by matching the country part and language part of the user's locale. If the country part does not exist, the infrastructure will attempt to match the language part only, and if this does not exist it will fall back to a default language. The localization of display names is illustrated below.

If the user is associated with the locale fr_CA, then the application searches the `MergedDefaultPreferences_<locale>.properties` files for the display names in the following order:

```
1) MergedDefaultPreferences_fr_CA.properties
2) MergedDefaultPreferences_fr.properties
3) MergedDefaultPreferences_en.properties
4) MergedDefaultPreferences.properties
```

The system first attempts to locate the correct display name for the `fr_CA` locale in a MergedDefaultPreferences_fr_CA.properties file. If this file does not exist, or if the display name for the user preference does not exist within this file, then the system looks for MergedDefaultPreferences_fr.properties. If this file does not exist, then the system will search for a `MergedDefaultPreferences_en.properties` file where locale is set to the default system locale. If the display name is not present the system will fall back to the `MergedDefaultPreferences.properties` file.

In the case where the display name is not found in any of the properties files (or the properties files do not exist), the value defined for the `name` attribute for a user preference in the DefaultPreferences.xml file will be used as the display name. See Section 13.2, *User Preferences Definition* for more details on the `name` attribute.

Similarly, if the user is associated with the locale en_US, then the application searches for the display names in `MergedDefaultPreferences_<locale>.properties` files with the following priority.

```
1) MergedDefaultPreferences_en_US.properties
2) MergedDefaultPreferences_en.properties
3) MergedDefaultPreferences.properties
```

13.6 Localizing Infrastructure Preferences Display Names

There are a number of Infrastructure Preferences used in the application and their display names can be localized in a manner similar to User Preference's display names. Localized display names can be added by creating new `InfrastructurePreferences_<locale>.properties`

files under the directory `EJBServer\components\<component_name>\userpreferences`.
Where `<locale>` represents the intended locale of the properties file and `<component_name>` is the name of a component within the component directory. A sample file, containing all the properties available for localisation, can be found in `SDEJ\lib\InfrastructurePreferences.properties`.

Notes

[1]See Section 10.2.2, *Customizing a Message File*, for further explanation of `SERVER_COMPONENT_ORDER`.
[2]See Section 10.2.1, *The Format of Message Files*, for further explanation of `SERVER_LOCALE_LIST`.

Chapter 14

Transaction Control

14.1 Overview

The *IBM Cúram Social Program Management* Server Development Environment (SDEJ) abstracts transaction management from the average developer. This section provides a brief overview for the developer and then details what is happening "under the hood". This is a difficult task because of multiple database support, which provide significantly different ways of supporting the *ACID* nature of a transaction. A transaction should be Atomic[1], its result should be Consistent[2], Isolated[3] and Durable[4].

14.2 Developer's View

14.2.1 Transactions and Method Invocations

Typically in Cúram a Facade method invocation maps to a single transaction. The exception to this is where the method invokes a deferred process or workflow. See the *Cúram Workflow Management System Developers Guide* for more details. The single transaction starts at the beginning of the Facade method invocation and finishes at the end.

The transaction demarcation in Cúram is bean managed rather than container managed and as such the developer must use the APIs in the infrastructure to checkpoint transactions.

One exception to this general rule is the Key Server. When a Unique ID block is obtained from the Key Server a separate transaction is started to govern this database access. This guarantees that long running transactions do not place locks on the Key Server tables as this would provide an unacceptable bottleneck.

14.2.2 Optimistic Locking and the forUpdate Flag

When a developer creates operations on an entity they must first determine if that entity supports optimistic locking. Optimistic locking is described in the *Cúram Modeling Reference Guide* and provides a suitable method of ensuring that transactions are ACID. However there are situations when using optimistic locking can unnecessarily impact on the performance of a transaction. If a record is read and then modified later in the transaction it is unlikely (though not impossible) that the record will have changed underneath the developer. Rather than using the version number it is often more suitable to lock the record when it is read. This means that it is impossible for another transaction to change the record, so there is no need to guard the modify with a version number. However it also means that the possibility of locks and deadlocks increases.

This form of locking is supported in Cúram via an extra parameter which can be passed to any of the standard read operations. This parameter (forUpdate), when set to `true`, will result in an update lock being taken on the record(s) that are being accessed as part of this query. These locks will not be released until the end of a transaction.

14.2.3 General Guidelines

There is a golden rule relating to locking and performance in database transactions. Any records you lock should remain locked for the minimum possible period of time to reduce database contention caused by other users seeking the same records. This means that operations that take out locks should be called as late as possible in your transactions. For example, if you read several records to validate a transaction, followed by updates to several more records, always perform the validations first followed by the updates. Try to defer update operations (or reads with locks) until as late as possible: don't scan a million-record table after taking out a record lock that ought to be short-lived.

14.3 Underlying Design

Transaction management happens on the server, rather than the client side. Client-initiated transactions would involve complicated and largely unnecessary communication overhead. However, this imposes a requirement on the application to guarantee that the database data remains consistent across a series of client/server calls. In practice this usually involves deferring the database updates done by a business function until the last client/server interaction in a series.

Transactions typically have to encompass interactions with more than one resource manager even if legacy systems are not used. The server database is one resource manager and the queues used for deferred processing and workflow are another. In order to guarantee atomicity of a transaction that is distributed across multiple resource managers, a two-phase-commit protocol is required to coordinate the distributed transaction.

14.3.1 DB2

At the beginning of a transaction Cúram obtains a single connection to the database. This connection runs at a specific isolation level:

- Repeatable Read - This guarantees that dirty data is not read and that a second read will read the same thing as a first.

However specific categories of statements are run at a lower isolation level:

- Cursor Stability - Cursor stability is the *DB2* implementation of the SQL standard `Read Committed` isolation level. This guarantees that a transaction cannot read a row with uncommitted changes in it. However it does not guarantee that a second read will read the same thing as a first.

This is not a separate connection to the database rather the *DB2* keyword `WITH CS` is automatically appended to the `SELECT` statement.

All queries which do not have the `forUpdate` flag set run at the "Cursor Stability" isolation level. All modifies and queries with the forUpdate flag set run at "repeatable read" isolation level. This means that they place a lock on the row(s) that have been read so that they cannot be updated by anyone else, and in the case of modify operations be read by anyone else. This lock is not released until the transaction commits.

14.3.2 Oracle

Oracle does not really support the JDBC Isolation levels (mainly because its underlying support does not truly map to these levels). For this reason *Oracle*'s default isolation level is used for all statements. In *Oracle* there is no possibility of a dirty read occurring.

Notes

[1]Atomicity requires that all of the operations of a transaction are performed successfully for the transaction to be considered complete. If all of a transaction's operations cannot be performed, then none of them may be performed.

[2]Consistency refers to data consistency. A transaction must transition the data from one consistent state to another. The transaction must preserve the data's semantic and physical integrity.

[3]Isolation requires that each transaction appear to be the only transaction currently manipulating the data. Other transactions may run concurrently. However, a transaction should not see the intermediate data manipulations of other transactions until and unless they successfully complete and commit their work. Because of interdependencies among updates, a transaction might get an inconsistent view of the database were it to see just a subset of another transaction's updates. Isolation protects a transaction from this sort of data inconsistency.

[4]Durability means that updates made by committed transactions persist in the database regardless of failures that occur after the commit operation and it also ensures that databases can be recovered after a system or media failure.

Chapter 15

Transaction SQL Query Cache

15.1 Overview

Benchmarking has identified that the same database query is often performed numerous times during one transaction in an *IBM Cúram Social Program Management* application. This is costly in performance terms and to counteract this a transaction SQL query cache is now available in the Server Development Environment (SDEJ). This cache, when enabled, operates at the data access layer and endures for the lifetime of any one transaction. The cache stores the results of any SELECT SQL queries for the duration of the transaction in which the operation was invoked. Subsequent calls in the same transaction will retrieve the required results from the SQL query cache and will not read the results from the database.

15.2 Populating the Cache

The SQL query cache will store the results in memory of any SQL query that executes a SELECT statement on a database table. Invocation of the following entity operation stereotypes will result in the results of that query being stored in the cache:

- <<read>>
- <<nsread>>
- <<nkread>>
- <<readmulti>>
- <<nsreadmulti>>
- <<nkreadmulti>>
- <<nsmulti>>

- `<<ns>>` with handcrafted SQL containing a SELECT statement

There are two exceptions to this rule:

- SQL queries that have the `FOR UPDATE` flag set to true will not have their results cached. These queries will always result in direct database access. This is due to the fact that this data is being read for modification and the subsequent update operation will result in that cache entry being invalidated.

- The results of specialized `<<readmulti>>` operations, where the operation is not an instance of `StandardReadMultiOperation` class, will *not* be cached. This is due to the fact that a customized `ReadMultiOperation` can modify the result set for a SQL query being executed. Since these results are not yet present in the cache, the cache cannot be invalidated which results in invalid data in the cache (i.e. the data cached for the SQL query does not reflect the data for that SQL query on the database).

15.3 Invalidating the Cache

The SQL query cache is associated with a transaction and is not global. When any specified transaction is committed or rolled back, the SQL query cache associated with that transaction is invalidated.

Any time an update (i.e. an insert, modify or remove operation) is made to a table associated with a transaction SQL query cache entry, that entry is invalidated from the cache. For most update operations (i.e. `<<modify>>`, `<<nsmodify>>`, `<<remove>>` etc.), the invalidation of cache entries is somewhat intelligent. The table affected by the update is determined from the SQL statement being executed and used to directly invalidate only the cache entries relating to the table. However, for `<<ns>>` operations that are executed and contain anything other than a SELECT SQL statement, the complete SQL query cache associated with that transaction is invalidated.

The following entity operations therefore cause the cache entries containing the table affected by that operation to be invalidated:

- `<<insert>>`
- `<<nsinsert>>`
- `<<modify>>`
- `<<nsmodify>>`
- `<<nkmodify>>`
- `<<remove>>`
- `<<nsremove>>`
- `<<nkremove>>`

- <<ns>> operation with handcrafted SQL that does not contain a SE-LECT statement

- <<batchinsert>>

- <<batchmodify>>

As detailed above the transaction SQL query cache endures for the lifetime of a transaction only. Database updates will result in the invalidation of associated entries in the local transaction cache only. As a result, any subsequent reads within a different transaction will return data from the cache and not as updated on the database.

15.4 Properties

The transaction SQL cache is enabled by default, meaning that the results of SQL queries *will* be cached. To disable it, the `curam.transaction.sqlquerycache.disabled` property must be set to `true` in the `Application.prx` file.

Storing the results of SQL queries that return large result sets may lead to memory problems in transactions that endure for a long period of time. The most likely queries that could lead to such problems are those that return data of type `CLOB` and `BLOB`. To cater for SQL queries that return large result sets, a property is available to control the size of fields of type `CLOB` or `BLOB` that may be stored in the transaction SQL query cache. This property is called `curam.sqlquerycache.lob.max.size` and it's default size is set to 500KB.

Further details concerning these properties may be found in Appendix A, *Cúram Configuration Parameters*.

15.5 SQLQueryCacheAdmin API

A public API is available for the transaction SQL query cache. The class, `curam.util.transaction.SQLQueryCacheAdmin`, provides functions that allow developers to manipulate the transaction SQL query cache at runtime. These methods include the following:

- *enableSQLQueryCache():* this function enables the SQL query cache for the current transaction.

- *disableSQLQueryCache():* this function disables the SQL query cache for the current transaction.

- *clearSQLQueryCacheForTable(String tableName):* this function flushes all entries from the transaction SQL cache that contain the specified table name for the current transaction.

- *clearSQLQueryCache():* this function flushes all of the entries from the transaction SQL cache for the current transaction.

15.6 SQLQueryCacheUtil API

A public API is available which contains utility methods for the transaction SQL query cache. The class, `curam.util.transaction.SQLQueryCacheUtil`, provides utility methods for the transaction SQL query cache. These methods include the following:

- *isSQLQueryCacheEnabled():* This function returns a flag to indicate if the transaction SQL query cache has been enabled or not.

- *runWithSQLQueryCacheDisabled(Runnable run):* This function runs the runnable bypassing the SQL query cache. SQLQueryCache may be needed to be disabled when there is a need to read the same row multiple times in a transaction to see if it has changed. For example, in the batch infrastructure it is required to read the same row multiple times in a transaction to see if it has changed.

15.7 Logging

When the tracing level for the Cúram application is set to `curam.util.resources.Trace.kTraceUltraVerbose` (see Section 8.4, *Logging Level* for more details on logging), various lifecycle events concerning the transaction SQL query cache are logged. These entries may be diagnosed in the logs by the following starting statement: `Transaction SQL Query Cache:`. The following events are logged during the lifecycle of the SQL query cache:

- When an entry is added to the transaction SQL query cache.

- When an entry is invalidated from the transaction SQL query cache.

- When the complete SQL query cache is invalidated as a result of a transaction being either committed or rolled back.

Chapter 16

Deferred Processing

16.1 Objective

In this chapter you will learn how to achieve deferred processing for appointed Business Process Objects (BPOs) in your *IBM Cúram Social Program Management* application.

16.2 Prerequisites

Before reading this chapter you should be familiar the *Cúram Modeling Reference Guide* and the Server Development Environment (SDEJ).

16.3 Introduction

In Cúram, describing a Business Process method as a Deferred Process means that this method is invoked asynchronously. A BPO within your Cúram application that calls a method of another BPO, configured for deferred processing, does not wait for it to return. Deferred Processing is accomplished, in part, by configuring queues in the middleware[1]. Any request over the queued enactment interface is deferred.

The structure of this guide is a step-by-step walk-through and explanation of what you must do in order to achieve deferred processing in your application.

16.4 Model Your Deferred Processes

A *deferred process* is identified in your application model by selecting the `<<wmdpactivity>>` stereotype on a method of a `<<process>>` class. Each deferred processing method must be defined to take the following input parameters:

> **Note**
>
> The application does not invoke a deferred process method using these parameters. These are the parameters passed to the method by the deferred processing server once the process is enacted.

- The ticket ID of the `DPTicket` record generated by the deferred processing engine (*long* datatype).

- The instance data ID (type of long) of the `WMInstanceData` record associated with the deferred process method at the time of enactment. This gives the deferred process method access to any information you wish it to have in order to carry out the required processing (*long* datatype).

- A boolean flag. This parameter is internal to the deferred processing infrastructure. It should be ignored, but *must* be part of the signature of the method (*boolean* datatype).

```
public void sampleDeferredMethod(long ticketID,
                                 long instDataID,
                                 boolean flag)
{
  // Method logic goes here
}
```

Example 16.1 <<wmdpactivity>> stereotype method

Example 16.1, *<<wmdpactivity>> stereotype method* shows the code that is generated for a method that is stereotyped as `<<wmdpactivity>>`. The required parameters must be specified in the model by the developer. You are not concerned with how these parameters are provided to the deferred process (that is taken care internally by the deferred processing engine following the enactment request). You, however, must code the logic of your deferred process into this method.

> **Note**
>
> Your deferred process should not attempt to perform any begin, commits or rollbacks via the `TransactionInfo` class or attempt any other forms of *Java EE* Transactional Control. This also applies to any methods that are invoked by workflows or deferred processes, regardless of how deep in the call stack. As well as deferred processes the examples of the methods include implementations of workflow or deferred processing interfaces (such as `Notifica-tionDelivery`, `WorkResolver`, `DPCallback`, etc.) and any methods called by either of the above.

16.5 Deferred Process Enactment

Deferred processes are enacted via the Deferred Processing Enactment Service.

Consider the situation where a BPO within your Cúram application needs to call a deferred process in order for it to do some other processing. The call must be made as shown in Example 16.2, *Using DeferredProcessing.startProcess*. Within the calling BPO you should populate a `WMInstanceData` record (see Section 16.5.1, *WMInstanceData*, how to define this entity) with the information that you want to be accessible to the deferred process.

The class `DeferredProcessing` is available to you from the SDEJ.

```java
import curam.util.AppException;
import curam.core.fact.WMInstanceDataFactory;
import curam.core.intf.WMInstanceData;
import curam.core.struct.UsersDtls;
import curam.core.struct.WMInstanceDataDtls;
import curam.util.fact.DeferredProcessingFactory;
import curam.util.intf.DeferredProcessing;
import curam.util.resources.GeneralConstants;
import curam.util.resources.KeySet;
import curam.util.type.UniqueID;

public class MyBPO extends curam.core.base.MyBPO {

  public void doOnlineOperation(int caseID,
                                UsersDtls usersDtls)
                                throws AppException {

    DeferredProcessing deferredProcessingObj
      = DeferredProcessingFactory.newInstance();
    WMInstanceData wmInstanceDataObj=
      WMInstanceDataFactory.newInstance();

    WMInstanceDataDtls wmInstanceDataDtls
      = new WMInstanceDataDtls();

    // Create a new instance data record
    wmInstanceDataDtls.wmInstDataID
      = UniqueID.nextUniqueID(KeySet.kKeySetDefault);
    wmInstanceDataDtls.caseID = caseID;
    wmInstanceDataDtls.enteredByID = usersDtls.userName;
    wmInstanceDataDtls.enteredByName = usersDtls.firstName
                                    + GeneralConstants.kSpace
                                    + usersDtls.surname;
    wmInstanceDataObj.insert(wmInstanceDataDtls);
    deferredProcessingObj.startProcess(
                        "DO_DEFERRED_OPERATION",
                        wmInstanceDataDtls.wmInstDataID);
}
```

Example 16.2 Using DeferredProcessing.startProcess

Example 16.2, *Using DeferredProcessing.startProcess* shows a Cúram application BPO that calls a deferred process method. The key points to note, however, are that the `WMInstanceData` record is set up as part of the calling BPO implementation. The `DeferredProcessing.startProcess()` is then used to request the enactment of the deferred process method. The parameters of this method are:

1. The name of the deferred process method being requested. This string value is configured by you in the `DPProcess` table. The exact configuration of the `DPProcess` table for deferred processing is dealt with in Section 16.7, *Configuration of Deferred Processing Table*.

2. The instance data ID of the `WMInstanceData` record that is populated with information that you deem necessary to be used by the deferred process.

3. `Optional`The Error Handler that implements the `TicketCall-back` interface that should be invoked if an error occurs. If the parameter is not provided the global error handler set through the property `curam.custom.workflow.ticketcallback` is called.

16.5.1 WMInstanceData

`WMInstanceData` allows the definition of application data that is particular to each deferred process, so that values can be supplied for that data for each instance of the deferred process.

Consider the situation where you want to develop a deferred method for processing a *Case*. The deferred processing engine has no knowledge of any cases (or even what a *case* is), so it cannot supply the ID of the *case* to your deferred method. It does, however, know about `WMInstanceData` and supplies the ID of a `WMInstanceData` record to every deferred method it invokes. This record should be created and populated by you before enacting the deferred process and the ID of the populated record should then be supplied to the enactment API. When the deferred processing engine invokes your deferred method, it will pass in that ID as a parameter.

Table 16.1, *WMInstanceData Properties* shows the `WMInstanceData` entity class and its properties. As you can see, apart from the unique identifier attribute of this class, all other information must be defined by you. This is done using the modeling environment. The `WMInstanceData` entity should be created in your model, in a package of your choice. `WMInstanceData` facilitates in the definition of your application specific information.

Property	Description	Type	Requirement
wmInstDataID	The unique identifier of the instance data.	WM_INST_DATA_ID	M
myInstanceData1	Property to be included as instance data	Your application domain definition for the property.	O
myInstanceData2 etc.	Property to be included as instance data	Your application domain definition for the property.	O

Table 16.1 WMInstanceData Properties

16.6 Offline Unit-Testing of Deferred Processes

If the application is deployed in an Application Server, the deferred methods will be invoked asynchronously. However, if the Application is *not* executing in an Application Server container (for example, for off-line unit-testing), you may wish to invoke the deferred method synchronously (i.e. not deferred). This can be done by setting the property `curam.test.stubdeferredprocessing` to `true`.

> **i** Note
>
> The invocation of the deferred method is dependent on a successful commit of the the caller's transaction context. If the calling method's transaction rolls back, the deferred process will not be invoked.
>
> Setting `curam.test.stubdeferredprocessinsametransaction` property to `true` ensures that the deferred processes gets invoked in the same transaction. If this property is not set, the deferred processes will still be invoked, but in a different transaction.

16.7 Configuration of Deferred Processing Table

When using deferred processing functionality in your Cúram application, you need to configure the `DPProcess` table prior to runtime in order for it to work correctly.

The `DPProcess` table, provided as part of the SDEJ, must contain names for the methods within your application that have been modeled and defined as being deferred using the `<<wmdpactivity>>` stereotype. For each deferred method, you must define a name that describes it, for the `processName` field. This string value is what must be used when requesting for a deferred process method to be enacted. The primary key of this table is a processName field.

Table 16.2, *DPProcess Properties* details the properties of the `DPProcess` table.

Property	Description	Type	Requirement
processName	Name that describes your deferred processing method.	String	M
interfaceName	Fully-qualified interface name of a BPO with a `<<wmdpactivity>>` method corresponding to the deferred process.	String	M
methodName	The name of the `<<wmdpactivity>>` method corresponding to the deferred process.	String	M

Property	Description	Type	Require-ment
ticketType	Code table value describing the type of deferred process. The meaning of this is Application-defined, for example, see the Cúram `TicketType` code table.	String	O
subject	Short description of what the deferred process method does.	String	O

Table 16.2 DPProcess Properties

Table 16.3, *Example DPProcess Table* shows an example of how a `DPProcess` table might be populated.

process-Name	interface-Name	method-Name	ticketType	Subject
DO_DEFERRED_OPERATION	server.curam.bizinterface.SomeProcess	doSomething	CLAIM	This method does something.
DO_ANOTHER_DEFERRED_OPERATION	server.curam.bizinterface.SomeOtherProcess	doSomethingElse	CASE-REVIEW	This method does something else.

Table 16.3 Example DPProcess Table

16.8 TicketCallback.dpHandleError()

The Deferred Processing Engine provides an error handling callback mechanism for when deferred processes fail (i.e. the deferred method you defined throws an exception). The `DPCallback` interface is provided with the infrastructure. It has a single method definition: `dpHandleError`.

`dpHandleError()` gives application developers control over error handling when the invocation of a deferred process fails. This callback is invoked once the deferred processing message has been moved to the `DPError` queue (usually after the failing process has been retried several times). An implementation example is provided in Example 16.3, *TicketCallback.dpHandleError()* below.

There are two ways an error handler can be configured. Firstly, a single

(global) error handler callback can be defined for deferred processing by setting the `curam.custom.deferredprocessing.dpcallback` property to the fully- qualified name of a class that implements the `DP-Callback` interface. The `dpHandleError()` method on that class will then be invoked when any deferred method fails. Alternately, you can supply the fully-qualified name of any class that implements the `DPCallback` interface when enacting a deferred process. This allows you to specify a specific error handler for a single deferred process, or even a subset of the instances a deferred process.

```
void dpHandleError(String processName, long instDataID)
  throws AppException {
  // Method logic goes here
}
```

Example 16.3 TicketCallback.dpHandleError()

This callback operation could be used to:

- Notify the client that a deferred process failed.

- Take some remedial action.

16.9 Security

Deferred processes run under the username 'SYSTEM'; therefore the effective locale for deferred processes is the default locale for this user as specified in field 'defaultLocale' on the Users table.

In the case of offline unit-testing of deferred processes, the username is blank and the effective locale is the default locale for the Cúram server.

16.10 Summary

- The incorporation of Deferred Processing into your application is largely achieved by:

  1. Modeling appointed BPO methods with `<<wmdpactivity>>` stereotype;

  2. Configuring the `DPProcess` table in your database;

  3. Using the `DeferredProcessing` to request deferred process methods.

- The appropriate deferred processing queues must be set up prior to runtime by following the steps given in the *Cúram Installation Guide*[2].

- Application specific error handling can be achieved using the `Ticket-Callback.dpHandleError()` method. An error handler can then be targeted in the code by passing the error handler class name when in-

voking the `DeferredProcessing.startProcess()` method.

Notes

[1]*WebSphere Application Server* and *WebLogic*. For exact details on the versions of these products, see the *Cúram Supported Prerequisites* document.
[2]You should refer to the installation guide for your particular platform type, i.e. *Windows* or *UNIX*.

Chapter 17

Timer Bean

17.1 Overview

Generic EJB Timer Bean functionality is provided as part of *IBM Cúram Social Program Management*, which allows users to start timers which will invoke client-visible methods at a specified point in the future either once or multiple times. This is based on the timer service provided by the EJB container. This chapter gives details about all major aspects of implementing Cúram Generic EJB Timer Bean.

17.2 EJB Timer Bean Definition

The EJB container provides the timer service, which is the infrastructure for the registration and callbacks of timers and, hence, provides the methods for creating and canceling them. The timer service of the enterprise bean container enables you to schedule timed notifications for all types of enterprise beans except for stateful session beans. You can schedule a timed notification to occur at a specific time, after duration of time, or at timed intervals. For example, you could set timers to go off at 10:30 AM on May 23, in 30 days, or every 12 hours.

The EJB container provides different types of timers. The timer can be a single-event timer, which can occur at a specific time or after a specific elapsed duration, or an interval timer, which may occur on a regular schedule. Essentially, three types of timers are possible, as outlined in the table below:

Type of Timer	Description
Single-event timer	Create a single-action timer that expires after a specified duration.
Single event with expiration date	Create a single-action timer that expires at a given point in time.

Type of Timer	Description
Interval timer with initial expiration Duration	Create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval.
Interval timer with initial expiration Date	Create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval.

Table 17.1 Types of Timers

17.3 Development Support

The Cúram infrastructure provides the following classes and interface to develop Timer Bean functionality.

- `curam.util.transaction.TimerInfo`

- `curam.util.timer.TimerTask`

- `curam.util.timer.TimerCallback`

17.3.1 **TimerInfo** Class

The class `curam.util.transaction.TimerInfo` contains methods for starting and stopping timers. This class also contains a number of internal methods and methods reserved for future use. The following table describes the API's that are currently supported by the infrastructure:

Method Name	Description
`startTask(long, TimerTask)`	Create a single-action timer that expires after a specified duration.
`startTask(long, long, TimerTask)`	Create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval.
`start-Task(DateTime, TimerTask)`	Create a single-action timer that expires at a given point in time.
`start-Task(DateTime, long, TimerTask)`	Create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval.
`cancel()`	Cancels the timer which invoked the current method. Should only be called by methods

Method Name	Description
	which were invoked by a timer, calling this method from a non-timed method will have no effect.
getID()	Gets the identifier for the timer which is running the current thread.
isTimerTransaction()	Indicates whether the current transaction is being run by a timer.

Table 17.2 List of API's in **TimerInfo** Class

17.3.2 **TimerTask** Class

The class curam.util.timer.TimerTask contains information about the timed operation, such as which server operation to invoke, parameters to pass into it, whether a callback is required, etc. The following table describes the parameters that are available in this class.

Name	Description
methodName	*Mandatory*. The name of the method to invoke when timer expires.
argument	*Optional*. A struct parameter for the method being invoked.
timerName	*Optional*. A name for this timer. This can be used as an identifier to query or cancel a timer.
errorHandlerName	*Optional*. The name of a class, which implements interface TimerCallback which will get called if the timed method fails.
userID	*Read-only*. The ID of the user who started off the task. This gets automatically populated when the timer is started.
taskID	*Read-only*. A unique identifier for each task. This is automatically populated when the timer is requested.
creationTime	*Read-only*. The time at which this timer was requested. This is automatically populated when the timer is requested.
initialDelay	*Read-only*. The initial delay time in milliseconds which was specified when this timer was created.
initialEventTime	*Read-only*. The absolute time of the first event for this timer, or null if none was specified when this timer was created.
Interval	*Read-only*. The repeat interval which was spe-

Name	Description
	cified when this timer was created, or zero if it is a one event timer.

Table 17.3 List of parameters from **TimerTask** Class

17.3.3 **TimerCallback** Interface

This is an interface for which developers can provide an implementation and which will get invoked if a timed operation fails. The interface `curam.util.timer.TimerCallback` has only one method `handleError(Exception, TimerTask)` defined and uses can provide implementation to this method.

17.3.4 Code sample:

```
// Create the task, specifying the name of the server
// operation to invoke:
final TimerTask task = new TimerTask();
task.methodName =
  "curam.core.facade.intf.ProductDelivery.close";

// This operation takes one struct parameter,
// so instantiate the struct and add it to the task:
final curam.core.facade.struct.CloseCaseDetails caseDetails
  = new curam.core.facade.struct.CloseCaseDetails();
caseDetails.caseID = 12345;
task.argument = caseDetails;

// Start off the timer, specifying that it invokes the
// method in 10 seconds time:
final long timerID = TimerInfo.startTask(10000, task);

// Every timer is assigned a unique ID which can be used
// to manipulate it and also to help track the timer
// in any diagnostic logs.
System.out.println("Created a timer with ID " + timerID);
```

17.4 Rules for using SDEJ Timers

There are some considerations and limitations to Generic Timer Bean provided as part of Cúram infrastructure and they are listed below.

1. SDEJ timers can invoke any client visible operation in the application meta-model, provided that:

   a. The operation has zero or one parameter.

   b. The operation has its Transactional option set to No.

   c. The user has access rights to that operation.

2. SDEJ timers do not have any facility to return a value from an operation.

3. Timer creation and cancellation are transactional; i.e., if you create a timer, it only becomes active after the transaction gets committed. Similarly if you cancel a timer, it only gets cancelled when that transaction gets committed.

4. Transactions invoked by timers execute using the same Cúram user ID as the user who created that timer.

5. The transaction type of a timer transaction is reported by `TransactionInfo.getTransactionType()` as being 'online'. (i.e. not deferred/batch/etc)

6. Timers should only be started by online transactions or other timer transactions. i.e. deferred processes, workflows or batch programs cannot start timers.

7. Timers are persistent and remain active until they are cancelled by the user, even if the application server is stopped and restarted.

8. If the application server is stopped for a time and then restarted later, all timers which were active before the shutdown will resume following the restart but the timer will not try to 'catch up' with any missed ticks. Instead it will tick at the next scheduled time.

9. If a timed operation throws an exception, the transaction will be rolled back. If the developer has specified a callback handler for the exception, the callback handler will get called if it has been configured, but it cannot be used to prevent the transaction from being rolled back.

10. If a timed operation throws an exception, the timer does not get cancelled and will continue to tick as before until it is cancelled from within a transaction which gets committed.

    Therefore it is important for developers to ensure that timed operations cannot repeatedly throw exceptions, as otherwise they could continue to be attempted indefinitely.

11. Timers should not be used to drive batch style processing. A timer driven transaction will have the same timeout as a deferred processing transaction (30 seconds by default) and should therefore be used only for reasonably short running pieces of processing.

12. Timers in the SDEJ are provided by the javax.ejb.TimerService of the application server. Currently it is not possible to start a timer from outside an application server which means that SDEJ timers are not available in the development environment. Attempting to start a timer from outside an application server will have no effect.

17.5 Timer Behavior

Timer can behave differently depending on the scenario at with they are started. Some of the scenarios and Timer behavior is as described below.

- For a repeating timer, if a timed transaction continues past the point at which the next tick is due, then that tick is discarded and the next due tick will be used.

  For example:

  A timer is configured to tick every 20 seconds. So this means that the timer will normally tick at the following times:

  20, 40, 60, 80, 100, etc

  Now let's say that on the second tick, the timed transaction took 25 seconds to complete. This means that the transaction which started at the 40 second mark completed at the 65 second mark, and is therefore deemed to have 'missed' the 60 second mark. So the next time the timer will tick will be at the 80 second mark. So the actual times the timer will have ticked are:

  20, 40, 80, 100

- When a timer is specified with an initial duration, that duration is relative to the time at which the timer was created. It is not relative to the time at which the transaction was committed - even though the timer cannot actually begin ticking until the transaction in which it was created has been committed.

  For example, the user invokes a rather long online transaction which does the following:

  - Creates Timer A with an initial duration of 60 seconds.

  - Does some processing which takes 20 seconds.

  - Creates timer B with an initial duration of 60 seconds.

  - Commits the transaction.

  Next the following will happen:

  - 60 seconds after it was created, Timer A will start ticking.

  - 20 seconds later, Timer B will start ticking.

  i.e. even though these timers were committed at the same time, each retains its own individual start time.

17.6 FAQ

- How do I see which timers are active?

  Different *Java EE* application servers implement their timer mechanism

in different ways and there is no standard way to administer timers via their admin consoles. The TimerInfo API provides a number of functions to find and query active timers.

- How do I stop a timer?

  A single-event timer will stop automatically after one successful execution. (i.e. if it executes a transaction which committed successfully.) For repeating timers, the TimerInfo class contains a number of methods for stopping these timers. Note that stopping a timer will only take effect when the transaction which requested the stop is committed.

- Can I ensure that my operation will be invoked only by a timer?

  Cúram timer beans can only invoke methods which are in the model and are client visible, therefore it is possible for the HTML client to also access these methods, which may not be desirable.

  If you want to ensure that only only a timer transaction executes your method, you can use the TimerInfo API to check for this at run time as illustrated by the following sample code extract:

```
// Ensure that this transaction is a timer:
if (!TimerInfo.isTimerTransaction()) {
 // throw an exception to report that an
 // invalid attempt was made to execute
 // this operation outside of a timer.
 throw new AppException(....);
}
```

- How many timers can be active at a time?

  The Cúram timer bean API is a wrapper for the *Java EE* Timer API and it is worth noting that the *Java EE* Timer API uses arrays of timers and as such is not designed for dealing with very large volumes of timers.

  As an extreme example: if an application contained several million customer records on the database, it would be unadvisable to use timers as the mechanism for controlling when an invoice is issued to each customer, because this would result in having several million timer objects active in memory.

  In general it is recommended that timers be kept as few and as short lived as possible.

- How accurate is a timer?

  The parameters used when creating a timer allow a developer to specify a granularity of milliseconds with regard to when and how often the timer will fire. However the application server cannot guarantee to fire the timer at exactly the expected time because there may be conditions which prevent this from being achieved. For example the server may be down at the scheduled time, it may be delayed by other transactions, a large number of timers may be scheduled to fire at exactly the same moment, etc. The rule of thumb is that the application server will fire the

timer event as close to the designated time as possible, so the developer should not assume that the timer will fire at an exact time.

- Can I use timers in the development environment?

  No. Currently timers only operate in deployed applications because the underlying implementation is provided by the application server.

- How can I debug timers?

  Timers cannot be executed in the development environment as this is currently not supported. However Cúram timers can output extensive logging data if required. The fact that each timer has a unique identifier means that its execution and life cycle can be traced via the log output.

  This logging data can be captured by configuring a *log4j* appender for category 'Trace.TimerInfo'.

- Can a timer be configured to start automatically?

  No. The life cycle of a timer is controlled by the developer. i.e. the developer is responsible for starting each timer and for ensuring that it stops.

Chapter 18

Events and Event Handlers

18.1 Overview

Events provide a mechanism for loosely-coupled parts of the *IBM Cúram Social Program Management* application to communicate information about state changes in the system. When one module in the application raises an event, one or more other modules receive notification of that event having occurred provided they are registered as listeners for that event.

To make use of this functionality, some events have to be defined, some application code must raise these events, and some event handlers have to be defined and registered as listeners to such events. Developers must write and register event handlers (classes that perform some action when an event is raised) and optionally event filters (logic that determines whether or not to invoke the handler for a given event). Event handlers and filters are classes that implement callback interfaces in much the same way as in the classic observer pattern [1].

18.2 The Format of Event Files

18.2.1 Event Definition

Events are defined in Cúram in XML files that specify both the event classes and the event types. These files are created with a `.evx` extension and are placed in the `events` of a Cúram component from where they are picked up and processed by the build scripts. The format of an event file is shown below:

```
<events package="curam.util.events">
  <event-class identifier="EVENT_CLASS_ONE" value="CLASS1">
    <annotation>Some event class.</annotation>
    <event-type identifier="EVENT_TYPE_ONE" value="EVENT1"/>
```

```
    <event-type identifier="EVENT_TYPE_TWO" value="EVENT2"/>
  </event-class>
  <event-class identifier="EVENT_CLASS_TWO" value="CLASS2">
    <event-type identifier="EVENT_TYPE_ONE" value="EVENT1">
     <annotation>Some event type.</annotation>
    </event-type>
    <event-type identifier="EVENT_TYPE_TWO" value="EVENT2"/>
    <event-type identifier="EVENT_TYPE_THREE" value="EVENT3"/>
  </event-class>
</events>
```

Example 18.1 Event definition file

events

This is the root tag of an event definition file under which all the event classes and types are defined.

package

This specifies the *Java* code package into which the *Java* constants for event classes and their types are generated.

annotation

This is an optional element specified for both event classes and types intended for descriptive text for the element. The text specified in an annotation is generated into the *Java* constant files as **javadoc** comments.

event-class

Defines an event class, which qualifies all the event types associated with that class.

identifier

This is the identifier of the event class for code generation and will be the class name for the constant class containing all the event types in the class. Since this will be a *Java* class name it must be a valid *Java* identifier.

value

This is how an event class is referenced at runtime and it is this value that event handlers are registered against. This value should be unique in the system and is a 100 character string.

event-type

Defines an event type within a given class. Since an event is identified by it's own name and that of it's parent class, an event type only needs to be unique within a given class.

identifier

This is the identifier of the event type for code generation and will be the field name for the constant containing the value of the event type. Since this will be a *Java* field name it must be a valid *Java* identifier.

value

>This is how an event type is referenced at runtime and the value should be unique within a given event class and is a 100 character string.

18.2.2 Event Handler Registration

Event handlers and their associated (optional) filters have to be registered against a particular event class to be invoked when the an event of the specified class is raised. This is done in file named `handler_config.xml` placed in the `events` folder of a Cúram component.

```
<registrations>
  <event-registration handler="curam.impl.SomeEventHandler">
    <event-classes>
      <event-class identifier="CLASS1"/>
    </event-classes>
  </event-registration>
  <event-registration handler="curam.impl.AnotherEventHandler"
                      filter="curam.impl.AnotherEventFilter">
    <event-classes>
      <event-class identifier="CLASS2"/>
    </event-classes>
  </event-registration>
  <event-registration handler="curam.impl.RemovedEventHandler"
                      removed="true">
    <event-classes>
      <event-class identifier="CLASS2"/>
    </event-classes>
  </event-registration>
</registrations>
```

Example 18.2 Event handler registration file

registrations

>This is the root tag of an event handler registration file under which individual registrations are defined.

event-registration

>Specifies an event handler registration.

handler

>The fully qualified name of an event handler class (see: Section 18.6, *Event handlers*).

filter

>The fully qualified name of an optional event filter class (see: Section 18.7, *Event filters*).

removed

>An optional attribute used by components of a higher precedence to disable previously registered event handlers, (see: Section 18.3.2, *Rules of Event Handler Merges*).

event-classes

This is a list of all the event classes against which the handler is registered.

event-class

A specific event class against which the specified handler is registered. When any event with the specified class is raised the event handler (providing the event filter approves) is invoked.

identifier

This identifies the event that the handler is registered against. This value should corresponds to the `value` attribute of an `event-class` element in the event definition files.

18.3 Merging Event Files

Both event definition and handler registration files are located in the `/events` directory of a component. The Cúram reference application is shipped with a set of event files. These may be augmented by placing new event files in the `SERVER_DIR/components/<custom>/events` directory, where `<custom>` is any new directory created under components that conforms to the same directory structure as `components/core`. This mechanism avoids the need to make changes directly to the out-of-the-box application, which would complicate later upgrades.

The override process involves merging all event files of the same name according to a precedence order. The order is based on the `SERVER_COMPONENT_ORDER` environment variable. This environment variable contains a comma-separated list of component names: the left most has the highest priority, and the right most the lowest.

After changing the component precedence order in `SERVER_COMPONENT_ORDER` it is necessary to perform a clean build to ensure that you are using the appropriate files. This is done by invoking **build clean server**.

18.3.1 Rules of Event Definition Merges

For event definitions to be merged, the files provided to customize the events must be named the same as the existing files containing the event classes to customize. Placing event classes with the same name in files with different names will result in errors occurring when loading the event definitions onto the database.

The customizing behavior for events is very simple; events cannot be removed as existing functionality might be using an event that other components then decide to remove. Such code would subsequently fail to compile. This being the case the only change that can be made to existing event definitions is that event types can be added to an event class by other components.

18.3.2 Rules of Event Handler Merges

The event handler (and filter) configurations used at runtime are those from the component with the highest precedence that specifies the event handler in question (for the purpose of merging the event handler is the identifier). Event classes that are to be processed by each handler as specified in the handler configuration in all the components are amalgamated into a merged configuration. It is also possible for higher precedence components to disable handler specified by lower precedence components by setting the `removed` attribute of the `event-registration` element to `true`.

18.4 Artefacts produced by generate events

There are two types of output generated by the evgen command: `.java` files (for code constants that make the use of events less error prone) and `.dmx` files (database scripts for loading event definitions onto the database).

The *Java* artefacts produced from a merged event files are placed in the `/build/svr/events/gen/[package]` directory. Where [package] is the `package` attribute specified in the event definition file. For example, package="curam.events" would result in the *Java* artefacts being placed in the `/build/svr/events/gen/curam/events` directory.

The database scripts produced from a merged event files are placed in the `/build/svr/events/gen/dmx` directory.

18.4.1 Database Scripts

Events are primarily a development time concept they are defined in XML files, raised in application code and handled by application defined call-backs. However some administration utilities in the application need access to the list of events defined and available in a running system; thus they are also loaded onto the data base.

Below are examples of the DMX files generated from the event definitions for the two entities used to store the event definitions.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<table name="EVENTCLASS">
  <column name="EVENTCLASS" type="text"/>
  <row>
    <attribute name="EVENTCLASS">
      <value>CLASS1</value>
    </attribute>
  </row>
  <row>
    <attribute name="EVENTCLASS">
      <value>CLASS2</value>
    </attribute>
  </row>
</table>
```

Example 18.3 Generated event class database script

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="EVENTTYPE">
  <column name="EVENTCLASS" type="text"/>
  <column name="EVENTTYPE" type="text"/>
  <row>
    <attribute name="EVENTCLASS">
      <value>CLASS1</value>
    </attribute>
    <attribute name="EVENTTYPE">
      <value>EVENT1</value>
    </attribute>
  </row>
  <row>
    <attribute name="EVENTCLASS">
      <value>CLASS2</value>
    </attribute>
    <attribute name="EVENTTYPE">
      <value>EVENT2</value>
    </attribute>
  </row>
</table>
```

Example 18.4 Generated event type database script

18.4.2 Java Code

Events are identified in the system by their names as specified by the `value` attribute of the `event-class` and `event-type` elements. However simply using text in application code to reference events would be error prone. In particular, an event is fully identified by its type as well as its class. Thus, using string literals to reference an event could be ambiguous, as an event type is only unique when qualified by its associated event class.

Below is an example of the generated constants file for an event class, the class name is the same as the event class, the attributes are the event types. This prevents the use of incompatible values.

```
package curam.util.testmodel.events;
/**
 * Generated EVENT_CLASS_ONE events file.
 * Some event class.
 *
 */
public final class EVENT_CLASS_ONE {

  /** Some event type. */
  public static final
    curam.util.events.struct.EventKey EVENT_TYPE_ONE
      = new curam.util.events.struct.EventKey();

  static {
    EVENT_TYPE_ONE.eventClass = "CLASS1";
    EVENT_TYPE_ONE.eventType = "EVENT1";
  }
```

```
  /** Another event type. */
  public static final
    curam.util.events.struct.EventKey EVENT_TYPE_TWO
      = new curam.util.events.struct.EventKey();

  static {
    EVENT_TYPE_TWO.eventClass = "CLASS1";
    EVENT_TYPE_TWO.eventType = "EVENT2";
  }
}
```

Example 18.5 Generated event Java constants

18.5 Raising events

Raising an event is simply a matter of creating an event struct, populating it with data, then calling the event service API to raise the event. The event infrastructure will notify any registered handlers that the event has been raised. An example of how to raise an event is shown below.

```
import curam.util.events.struct.Event;
import curam.util.events.impl.EventService;
curam.util.events.EVENT_CLASS_ONE;

...

Event event = new Event();
event.eventKey = EVENT_CLASS_ONE.EVENT_TYPE_TWO;
event.primaryEventData = 12300838;
event.secondaryEventData = 23413081;

EventService.raiseEvent(event);
```

Example 18.6 Raising an event

eventKey

> This is the unique identifier of the event within the system. It is made up of two constituent parts: the event class and the event type. As mentioned earlier and as shown in the example, though the event key is two parts it is best to specify it using one generated constant to avoid mismatching event classed and types.

> **eventClass**

>> The class of the event being raised: this is the value on which handlers are registered.

> **eventType**

>> The type of the event being raised: this identifies the specific type of the event in the given class.

primaryEventData

This is the primary payload of the event and is a 64-bit integer. Typically this will be (though not necessarily) the identifier of an entity in Cúram, the entity in question being identified by the class of the event. The event type is commonly used to indicate the action that has taken place on the entity.

secondaryEventData

This is any additional data that may be associated with an event when it is raised. Unlike the primary event data the secondary event data is optional.

18.6 Event handlers

We have already seen how to register handlers. To create an event handler one simply needs to implement the interface: `curam.util.events.impl.EventHandler`, which is shown below.

The action taken by an event handler when the event is raised is up to the developer. It should be noted that event handlers are invoked synchronously when the event is raised (and hence run within the same transaction context as the code raising the event). This has two implications:

- Throwing exceptions from an even handler will result in the transaction from which the event was raised being rolled back.

- Long running actions should be avoided in event handlers as they will affect the running time of the code raising the event.

```
package curam.util.events.impl;

import curam.util.events.struct.Event;
import curam.util.exception.AppException;
import curam.util.exception.InformationalException;

public interface EventHandler {
  void eventRaised(Event event)
    throws AppException, InformationalException;
}
```

Example 18.7 Event handler interface

18.7 Event filters

As mentioned previously, an event handler can be configured to have a filter. The purpose of a filter is to decide whether or not the handler needs be notified about the event being raised. To create an event filter one simply needs to implement the interface: `curam.util.events.impl.EventFilter` which is shown below.

If the `accept` method returns `true` the event will be passed on to the

event handler (that is the `eventRaised` method of the associated event handler will be invoked), otherwise the event is ignored.

```
package curam.util.events.impl;

import curam.util.events.struct.Event;
import curam.util.exception.AppException;
import curam.util.exception.InformationalException;

public interface EventFilter {
  boolean accept(Event event)
    throws AppException, InformationalException;
}
```

Example 18.8 Event filter interface

Notes

[1] The observer pattern is one of the design patterns made popular by the landmark book *Design Patterns: Elements of Reusable Object-Oriented Software*. It describes a generic listener framework.

Chapter 19

Unique IDs

19.1 Overview

This chapter describes what Unique IDs (Identifiers) are in the context of *IBM Cúram Social Program Management* and how to use them in your application.

19.2 What are Unique IDs?

Unique IDs are numbers generated by the Cúram infrastructure for use as unique database keys. They come in two flavors:

- *Human-readable Unique IDs* are ascending sequences of numbers, usually starting at 1, and are used as database keys where the key value might need to be presented in a User Interface to a human user.

- *Non-human-readable Unique IDs* are typically large positive or negative values in the approximate range 1E-19 to 1E+19. The sequence of non-human-readable Unique IDs does not repeat (for 2^64 key values), but is random in a way that can improve database performance in some circumstances.

A Unique ID key set is a named non-repeating set of 2^64 Unique ID key values. Key sets can be configured by developers and used to generate Unique IDs for a particular purpose. Each key set can be configured to be human-readable or non-human-readable. The infrastructure uses a number of predefined key sets which must be configured as part of a Cúram installation.

19.3 What are Unique IDs for?

Cúram-generated Unique IDs address a perennial problem in application

design - how to co-ordinate multiple processes each of which needs to allocate a number guaranteed to be unique throughout the application. One classic approach involved locking and updating a key control database table each time a key needs to be allocated. Unfortunately, this can lock the control table for the duration of long-lived transactions, preventing other processes from accessing it. This technique is almost always the source of serious database contention problems in an application (see "Allocating Sequence Numbers" from Chapter 12 of *High Performance Client/Server*, Loosley and Douglas).

Unique IDs are served out in blocks of 256 keys using a unique ID generator, also know as the *Key Server* [1]. A process requests a block of Unique IDs by calling the key server. This updates a database control table each time it returns a block of Unique IDs to a requesting process. Once a block has been allocated, the requesting process can allocate keys from this block locally, i.e. without calling the server again, until the Unique ID block is exhausted. Furthermore, the key server operates in its own transaction so it never locks the key control table for longer than it takes to allocate and update a next Unique ID block value.

It should be noted, however, that a process which requests a Unique ID block may or may not use the keys from that block. If it does not, then the unused keys represent holes in the key sequence. Processes which use, say, one key value before shutting down will leave quite large holes in the key sequence. Note also that there is no time limit on how long a process can wait between allocating a Unique ID block, and using the key values in it. Thus, even for human-readable keys which are in an ascending sequence starting at 1, the sort order of keys on the database has no direct bearing on the chronological order in which they were inserted. Obviously, programs should not assume that this is the case.

19.4 Can I run out of Unique IDs?

In short, no! A process which used only one key out of each Unique ID block, and allocated one thousand of these per second non-stop, would take over two million years to exhaust one Unique ID key set. For all practical purposes, the set of Unique IDs in a key set can be considered to be inexhaustible.

19.5 When should I use Unique IDs?

Use Unique IDs in your design when each of the following criteria are met:

• you need a unique key for a database entity

• the database key has no "business meaning"

• instances of the entity may be created by multiple contending on-line or batch functions

- holes in the key sequence are acceptable (which should always be true if the key has no business meaning).

19.6 When should I not use Unique IDs?

Do not use Unique IDs in your design when:

- you need a unique key for a database entity, but have a business requirement for an ascending sequence without holes (Cúram-generated Unique IDs are not guaranteed to be contiguous), or

- your key requires something other than a simple numeric format, or

- contending processes will not create instances of the entity (in which case there is no need for key control at all).

19.7 Should my keys be human-readable?

This is up to you. The rule of thumb is that Unique ID values that will be displayed to a user should be human-readable. Otherwise, you may choose to use non-human readable Unique IDs. The advantage of these is that their values are spread across a very large range, so that database indexes are not always being extended at the end, as for ascending sequences.

19.8 What if I require contiguous human-readable Unique IDs?

Human-readable IDs allocated by the key server are sequential, but can have gaps for two reasons:

- the IDs are allocated in blocks of 256 keys. When the server is restarted, the remaining values in any block for any key set that has been loaded are discarded.

- if a transaction that requests a human readable ID from the key server is rolled back, the ID that was served up is discarded (as the key server runs in a separate transaction, its transaction commits irrespective of what happens to the application transaction - this is important for performance reasons).

In instances where there is a requirement to generate human-readable IDs, where the numbers must be both sequential and have no gaps, Cúram uses an application defined "key" table for each set of IDs (for example, InternalPersonID, InternalEmployerID, etc). An example of such a business requirement is the one around the issuing of "Social Security Numbers". These tables are read and updated in the context of the application transaction, meaning, the ID is only allocated if the record bearing that ID is committed to the database. Otherwise, the whole business transaction, including

the ID allocation, is rolled back. It is worth noting that there is a performance overhead because of this, as the single row ID table is a database hot spot that must be updated every time the record bearing that ID is committed to the database.

Thus it is recommended that:

- this method of ID generation is used only when absolutely necessary and

- your design should strive to ensure that transactions using this mechanism are kept as short as possible to minimize contention on the key table.

19.9 How do I use Unique IDs?

Designing Unique IDs into your Cúram application is straightforward. In your UML application model, set the appropriate domain definitions to be of the data type `SVR_INT64`. The developer's view of this is as a *Java Long* primitive. To allocate a new Unique ID call `UniqueID.nextUniqueID()`, passing a key set name as a string. This call transparently looks after allocating a new Unique ID block if necessary. If no key set name is passed to the `nextUniqueID()` method the default key set, `curam.util.resources.KeySet.kKeySetDefault`, is used. This key set allocates non human-readable Unique IDs.

Key sets are defined by configuring entries in the `KeyServer` database table. This can be done by creating a DMX file defining all key entries. Table 19.1, *KeyServer Database Table* details the fields of the `KeyServer` database table.

Field	Description
keySetCode	An identifier for the key set, e.g. MYKEYSET.
nextUniqueIdBlock	The next Unique ID block that should be allocated. For human-readable IDs this field can be used to skip pre-allocated Unique IDs.
humanReadable	True if the Unique IDs should be human-readable.
lastUpdated	The timestamp for when the entry was last updated.
strategy	Represents strategy used to generate next unique id block for a given key set.
Annotation	A description of the key set.

Table 19.1 KeyServer Database Table

If you are using human-readable Unique IDs, and non-Cúram-generated keys have already been allocated, then you can guarantee that these values will never be re-allocated by Cúram (i.e. Unique IDs will never "clash"). This is achieved by setting the `nextUniqueIdBlock` field on the `KeyServer` database table to be *Ceiling(N/256)*, where N is the number of

Unique IDs which have already been pre-allocated.

The strategy field is used to specify whether the standard Key Server or the Range Aware Key Server is used for the key set. If the field is set to null, the standard Key Server is used. If the field is set to a specific value KB1002 then the Range Aware Key Server will be used to generate next unique id block for the key set. The Range Aware Key Server is explained in more detail in Section 19.10, *Range Aware Key Server*.

> **Warning**
>
> Care should be taken when defining and using custom key sets. The same key set should always be used when using Unique IDs as the primary key for a particular database table. If two key sets are used to generate Unique IDs for the same database table, duplicate record problems may occur. Unique IDs are only unique within a particular key set.

> **Note**
>
> The conversion routine for hexadecimal numbers that are used as Unique IDs on a *DB2 for z/OS* database can only support numbers between `Long.MAX_VALUE` and `Long.MIN_VALUE + 1`.

19.10 Range Aware Key Server

19.10.1 Overview

The Range Aware Key Server is a new Key Server implementation introduced to support Configuration Transport Manager (CTM). CTM is used to transport administrative configuration data (Business Objects) between systems. Each Business Object is comprised of a number of entities. Each of these entities has a primary key. The standard Key Server implementation only guarantees uniqueness of a primary key within a single system installation. This means that when a Business Object is transported from a Source System and applied on a Target System, there is the strong possibility of key clashes between the transported entities and the existing entities on the system.

The Range Aware Key Server implementation is responsible for creating primary keys to meet the following requirements:

- Prevent clashes in primary keys between new entities transported to a system and existing entities on that system.

- Identify where there is an existing version of a transported entity on a system, so that the existing entity is updated with the transported entity data.

19.10.2 How does the Range Aware Key Server work?

The approach used by the Range Aware Key Server to generate primary keys hinges on ensuring that non-overlapping key ranges are allocated to every system. The Range Aware Key Server will then ensure that all of the primary keys on a given system are generated from the range(s) assigned to that system. Therefore, the primary keys generated by each system will be unique.

So, at system install (or upgrade) time, the system administrator allocates a unique primary key range from which all primary keys provided by the Range Aware Key Server implementation will be generated. Please refer to the *CTM Setup Guide* chapter in the *Configuration Transport Manager* guide for information on how the range allocations are configured.

19.10.3 Where is the Range Aware Key Server used?

The Range Aware Key Server is only used for Key Sets that have been created specifically for the entities that form part of transportable Business Objects. Existing Key Sets continue to use the current SDEJ Key Server implementation, unchanged. Note that it is important that existing Key Sets are not changed to use the Range Aware Key Server - the Range Aware Key Server should only be used with new Key Sets.

The Range Aware Key Server supports both non-human readable and human-readable generated keys, so the value of the humanReadable attribute in the KeyServer table is set to either 0 or 1 depending on the entity's requirements.

Notes

[1] The design is loosely based on the Sequence Block pattern described by Floyd Marinescu in EJB Design Patterns (ISBN: 0471208310).

Appendix A

Cúram Configuration Parameters

A.1 Overview

This section describes configuration parameters for Curam applications that you can (or in some cases must) set to control characteristics of application execution. Generally, and unless otherwise noted, these parameters are set in property and prx files associated with your application. The following configuration parameter descriptions are organized according to the file that they should be set in and also in functionally-related groups. Some parameters are of a "BOOLEAN" type, where noted. This means that the value "true" or "yes" in upper-, lower-, or mixed-case, equates to a "true" value; all other values (or none) equate to "false". The configuration parameter descriptions are grouped into functionally-related groups.

A.2 Bootstrap.properties

The following properties relate to the Bootstrap.properties file.

A.2.1 Database

These settings configure Curam for database communication.

Property Name	Type	Meaning
curam.db.type	STRING	The property which specifies the database type. Suggested: DB2/ORA/ZOS.
curam.db.password	STRING	The encrypted password that corresponds to the user name specified above. The database password is never stored in plaintext in the various Curam property

Property Name	Type	Meaning
		files.
curam.db.username	STRING	A valid database username.
curam.db.oracle.cachesize	INT32	The size of the prepared statement cache used by batch programs when run against Oracle (the prepared statement cache is based around implicit caching).
curam.db.oracle.connectioncache.enabled	BOOLEAN	Turn on connection caching for Oracle outside of an Application Server.
curam.db.oracle.connectioncache.minlimit	INT32	Set Min Limit for the Cache. This sets the minimum number of PooledConnections that the cache maintains. This guarantees that the cache will not shrink below this minimum limit.
curam.db.oracle.connectioncache.maxlimit	INT32	Set Max Limit for the Cache. This sets the maximum number of PooledConnections the cache can hold. There is no default MaxLimit assumed meaning connections in the cache could reach as many as the database allows.
curam.db.oracle.connectioncache.initiallimit	INT32	Set the Initial Limit. This sets the size of the connection cache when the cache is initially created or reinitialized. When this property is set to a value greater than 0, then that number of connections are pre-created and are ready for use.
curam.db.oracle.connectioncache.name	STRING	The name used to identify the cache uniquely.
curam.db.zos.32ktablespace	STRING	Property which specifies the name of the table space used for 32k storage on DB2 z/OS.
curam.db.zos.enableforeignkeys	BOOLEAN	Controls whether foreign keys are generated for a z/OS database when running the Data Manager. Note on usage - If Foreign Keys are used against a z/OS database, the tables are put in a CHECK_PENDING state, causing failures when the tables are

Property Name	Type	Meaning
		accessed. The state can only be changed through direct DBA intervention on the target platform (hence it cannot be scripted into the Data Manager which can run on remote clients). In normal usage the Data Manager invokes LOB Manager after applying the foreign keys. This means the LOB manager should be re-run after the this CHECK_PENDING state has been resolved.
curam.db.disableforeignk eys	BOOLEAN	Controls whether foreign keys are generated in SQL statements. By default this property is false, which means foreign key generation is enabled. However, for z/OS foreign keys will not be generated if curam.db.zos.enableforeignkeys is set to false.
curam.db.disableInvalidL obFileError	BOOLEAN	This property controls the reporting of invalid LOB file paths in DMX files. The default value is FALSE. By default a build exception will be thrown, when set to TRUE a warning will be reported.
curam.db.zos.encoding	STRING	Property which specifies whether the database being used on z/OS requires processing for EBCDIC, ASCII, or UNICODE encoding. This should be set to EBCDIC, ASCII, or UNICODE depending on the appropriate database encoding in use. EBCDIC is the default value.
curam.db.zos.dbname	STRING	The name of the database on z/OS.
curam.database.shortnam es	BOOLEAN	It is strongly recommended that this property be set to false. The functionality for this property is planned for removal in a future version of Curam. If you have utilized this property in previous

Property Name	Type	Meaning
		versions of Curam please contact Curam Support for more information.
curam.db.oracle.servicename	STRING	The Oracle database service name. Setting this will create database connection using Oracle service name.
curam.db.name	STRING	The database name. This setting will be overridden if property "curam.db.oracle.servicename" is set for Oracle database.
curam.db.servername	STRING	The database server name.
curam.db.serverport	INT32	Suggested: 1521 (Oracle)/ 50000 (DB2). The database server TCP/IP port.
curam.db.enable.bindings.generation	BOOLEAN	Suggested: false. Causes a bindings file to be generated for the JDBC data source when a database connection is made outside of the application server, e.g. by the Batch Launcher. Has no effect if property 'curam.db.disable.bindings.generation' is set. Intended to be used to produce a starter bindings file which can then be customized.
curam.db.disable.bindings.generation	BOOLEAN	Suggested: false. Prevents re-generation of the JDBC data source bindings file and instead causes the data source to be looked up from a customized bindings file when a database connection is made outside of the application server, e.g. by the Batch Launcher.
curam.dmx.locale	STRING	Default: en. Property that specifies the locale that will be used when inserting dmx data onto the database. The locale should be specified in the format: language_country, for example en_US.
curam.db.multibyte.expansion	BOOLEAN	Enables the multi-byte expansion feature. Default value is true.

Property Name	Type	Meaning
curam.db.multibyte.default.factor	FLOAT	Specifies the default expansion factor for multi byte string fields if multi byte expansion is enabled. Must be a float between the values of 1 and 4. Default value is 4.

Table A.1 Database settings

A.2.2 Environment

These settings configure the environment for your Curam application.

Property Name	Type	Meaning
curam.environment.as.vendor	STRING	Suggested: Should be set to BEA or IBM to reflect the Application Server which is being used. If running outside an application server this should be empty. Defines the Application Server in which Curam will be deployed. This is setup automatically when the EAR file is built using the build targets.
curam.environment.tnameserv.port	INT32	Suggested: 900. Port on which the tnameserv is running.
curam.environment.bindings.location	STRING	Suggested: C:/Temp. Name of the file system location containing data sources.
curam.environment.default.dateformat	STRING	Default: yyyy MM dd. The date format. Can be set to one of: "d M yyyy", "M d yyyy", "yyyy M d", "dd MM yyyy", "MM dd yyyy", "yyyy MM dd", "d MMM yyyy", "MMM d yyyy", "yyyy MMM d", "d MMM yyyy", "MMMM d yyyy", "yyyy MMMM d", "dd MMM yyyy", "MMM dd yyyy", "yyyy MMM dd".
curam.environment.default.dateseparator	STRING	The date separator. Can be set to one of: ".", ",", "/", "-".
curam.disable.dynamic.properties	BOOLEAN	Indicates if dynamic properties should be enabled or disabled. This is used by command line

Property Name	Type	Meaning
		tools that require access to properties but cannot access the database.
curam.deprecation.reporting	BOOLEAN	Indicates if deprecation reporting should be enabled or disabled. This is used by all tools (both online and offline) that report deprecation warnings to the user (e.g. rules and workflow validation).
curam.entity.struct.deprecation	BOOLEAN	Indicates if generated entity standard structs should be deprecated if an entity is deprecated. This is used by generators which generate standard entity structs.
curam.environment.roundingprecision.enable	STRING	Indicates if when rounding money types in Curam, the HALF_UP algorithm will be used. This means that all Money will be rounded up. If set to true, the HALF_UP algorithm will be used. If not set, a default of true is used.

Table A.2 Environment settings

A.2.3 Test

These settings configure those elements of Curam which are useful for Unit Testing. None of these settings should be used in a deployed application as they will either degrade performance or cause failures.

Property Name	Type	Meaning
curam.test.override.date	STRING	This property allows the date and time to be set to a known value for testing. In order to override the date and time the property should be in the format YYYYMMDDThhmmss. The 'T' character is the separator between the date and the time. It is valid to only specify the date. If the time portion of the property is not explicitly set the time

Property Name	Type	Meaning
		will be automatically default to midnight (00:00:00). For example, the string value 20070101T175930 represents 17:59:30 on 1st January 2007. The string value 20070101 represents 00:00:00 on 1st January 2007.
curam.test.treatreadmulti maxaserror	BOOLEAN	Default: false. Specifies that a run time error should be thrown as well as a log message when the result size of Readmulti operation exceeds the maximum. This does not apply when the Treat readmulti-max as InformationalException option is enabled

Table A.3 Test settings

A.2.4 Custom

These settings allow a developer to replace elements of the Curam infrastructure with their own customized handlers.

Property Name	Type	Meaning
curam.custom.workflow. webservicebpo	STRING	The name of the application BPO that workflow process enactment web services go through.

Table A.4 Custom settings

A.3 Application.prx - Dynamic properties

The following properties relate to the available dynamic properties in the Application.prx file.

A.3.1 Environment

These settings configure the environment for your Curam application.

Property Name	Type	Meaning
curam.environment.default.locale	STRING	Default: en. The default value of the language code for the server.
curam.environment.recor	BOOLEAN	Specifies whether a RecordLock-

Property Name	Type	Meaning
dlocked.systemexception		edException should be set to a System exception. The default is false here, that it is a Application exception.
curam.environment.read mul- timax.systemexception	BOOLEAN	Specifies whether a Readmul- tiMaxException should be set to a System exception. The default is false here, that it is a Applica- tion exception.
curam.transaction.sqlquer ycache.disabled	BOOLEAN	Specifies whether any SQL quer- ies that do a SELECT on a data- base table will have their results cached for the duration of the transaction in which the opera- tion was invoked. Subsequent calls using the same SQL query will then retrieve the results from this thread local transaction SQL query cache and not read the res- ults from the database. The de- fault setting for disabling this cache is false so that the results of SQL queries will be cached.
curam.sqlquerycache.lob. max.size	INT64	Specifies the maximum size of a field of type CLOB or type BLOB in a result set that is al- lowed to be cached in the trans- action SQL query cache.
curam.enable.logging.clie nt.authcheck	BOOLEAN	Default: false. When set to true, all client authorization checks will be logged to the Authorisa- tionLog database table.
curam.audit.audittrail.dat acompressionthreshold	INT32	Specifies the size of the audit data stored in the detailinfo column of the audittrail database table that causes data compres- sion to be invoked. Default: -1 (off). This value is checked per audit operation. To turn compres- sion on for all audittrail detail- info data set this value to 0. When turned on rows that con- tain compressed data have the boolean attribute ISCOM- PRESSED set to true. Note that short audit data is not likely to

Property Name	Type	Meaning
		see performance gains, but will for large data rows. The performance of Curam auditing OOTB should not require compression, but if you add additional auditing you should evaluate your auditing selections for performance to determine the best setting for this value. Compression is done via the curam.util.resources.ByteArrayUtil.byteArrayToBase64EncodedString method and decompression can be done via the corresponding ByteArray-Util.base64EncodedStringToByteArray method.

Table A.5 Environment settings

A.3.2 JMX

These settings configure the JMX infrastructure for your Curam application.

Property Name	Type	Meaning
curam.jmx.monitoring_enabled	BOOLEAN	Whether JMX monitoring is enabled or not in the application.
curam.jmx.transaction_tracing_enabled	BOOLEAN	Whether transaction tracing is enabled or not in the application. When this is enabled, in-flight data collection is enabled as well.
curam.jmx.transaction_tracing_url_filter	STRING	Regular expression to identify URLs for which transaction tracing data is collected.
curam.jmx.transaction_tracing_max_recorded_threads	INT32	The maximum number of threads for which transaction tracing data is collected. Note that at any one moment there could be more than this number of threads in the transaction tracing data but a significant amount of entries will only be preserved for this number of threads.
curam.jmx.transaction_tracing_purge_period	INT32	The period of time, in seconds, between checks to ensure that

Property Name	Type	Meaning
		only the number of threads specified in curam.jmx.transaction_tracing_ max_recorded_threads are preserved in the transaction tracing data.
curam.jmx.transaction_tracing_max_thread_idle_time	INT32	The maximum amount of time, in seconds, a thread is allowed to be idle before its transaction tracing data can be cleared.
curam.jmx.configured_mbeans_ejb	STRING	The list of MBeans configured in the EJB container.
curam.jmx.configured_mbeans_web	STRING	The list of MBeans configured in the WEB container.
curam.jmx.per_user_statistics_filter	STRING	Regular expression to identify users for which individual statistics are collected.
curam.jmx.in_flight_statistics_enabled	BOOLEAN	Whether or not statistics about in-flight transactions are collected.
curam.jmx.sql_statement_statistics_enabled	BOOLEAN	Whether or not SQL statement statistics collection is enabled.
curam.jmx.download_statistics_allowed	BOOLEAN	Whether or not the download of JMX statistics is allowed.
curam.jmx.download_statistics_username	STRING	The username of the user who is allowed to download the JMX statistics.
curam.jmx.end_user_statistics_enabled	BOOLEAN	Whether or not end user statistics collection is enabled.
curam.jmx.end_user_statistics_user_filter	STRING	Regular expression that selects users for which end user statistics are collected.
curam.jmx.end_user_statistics_display_enabled	BOOLEAN	Whether or not the end user statistics are displayed in the browser. If true, the statistics for the current page are displayed in the top left corner of the page.
curam.jmx.end_user_statistics_upload_delay	INT32	The delay in seconds between the page reporting being loaded and the moment the statistics are uploaded.

Table A.6 JMX settings

A.3.3 Test

These settings configure those elements of Curam which are useful for Unit Testing. None of these settings should be used in a deployed application as they will either degrade performance or cause failures.

Property Name	Type	Meaning
curam.test.store.entitykeys	BOOLEAN	Default: false. Specifies that the values written to the database should be stored in memory for retrieval by tests. They can be accessed through curam.util.DataAccess.KeyRepository.
curam.test.trace.statistics	BOOLEAN	Default: false. Place a compact trace of BO method invocations in a buffered log. This representation is suitable for obtaining performance measurements.
curam.test.trace.statistics. location	STRING	The name of the file which will have the statistics information generated into it.
curam.test.singleuser	BOOLEAN	Indicates whether only a single user will be active. This is the only mode supported if an IDE is used to execute Curam as a standalone Java program.
curam.test.stubdeferredprocessing	BOOLEAN	Default: false. Specifies that it needs to use deferred processing without en-queuing in App Server.
curam.test.stubdeferredprocessinsametransaction	BOOLEAN	Default: false. Specifies that stubbed deferred process calls should be run in the current transaction using the current database connection. If true, a new transaction will not be started for each stubbed deferred process call.

Table A.7 Test settings

A.3.4 Rules

These settings configure the rules infrastructure of Curam.

Property Name	Type	Meaning
curam.rules.file.access.location	STRING	The directory where the XML representation of rule sets will be created.
curam.rules.file.access.multilocation	BOOLEAN	Specifies that rule set files exist in multiple locations.
curam.rules.model.file.rdo.access	BOOLEAN	Specifies that RDOs should be retrieved from a Curam model file.
curam.rules.default.locale	STRING	Default: en_US. Default locale used when creating the XML representation of rule sets.
curam.rules.globals.description	STRING	The display/user friendly name associated with the pre-defined Globals Rules Data Object. The default value is the localized message text associated with the infrastructure catalog entry: RULES:ID_GROUP_DISPLAY_NAME_GLOBALS
curam.rules.enable.optimization	BOOLEAN	Specifies the rules optimization.
curam.rules.enable.fulltext	BOOLEAN	Specifies the rules engine construction of full result text.
curam.debug.rules	BOOLEAN	Default: false. Specify whether the rules debugging should be enabled.
curam.disable.empty.objectivelistgroups	BOOLEAN	Default: true. Specify whether the rules decision should include empty Objective list groups.
curam.rules.date.range.includes.calculation.date	BOOLEAN	Specifies the new objective calculation.

Table A.8 Rules settings

A.3.5 IEG

These settings configure the properties which relate to the IEG Environment.

Property Name	Type	Meaning
curam.iegeditor.callback.class	STRING	Specifies the IEG Editor Application Callbacks class.
curam.iegruntime.questio	BOOLEAN	Specifies whether to use separate

Property Name	Type	Meaning
npage.separatequestionsf orloopstyle		question pages when "for" loop- ing.

Table A.9 IEG settings

A.3.6 Custom

These settings allow a developer to replace elements of the Curam infra- structure with their own customized handlers.

Property Name	Type	Meaning
curam.custom.deferredpr ocessing.dpcallback	STRING	The name of the application class that implements the DPTicket- Callback interface.
curam.custom.workflow. workresolver	STRING	The name of the application class that implements the WorkRe- solver interface.
curam.custom.workflow. processcachesize	INT32	Default: 250. Specifies the max- imum size of the process defini- tion cache.
curam.audit.audittrail.nox mlaudit	BOOLEAN	If set to true this property will disable the existing audit writer.
curam.custom.notificatio ns.notificationdelivery	STRING	Specifies the name of the applic- ation class that implements the NotificationDelivery interface.
curam.custom.dataaccess. databasewritecallback	STRING	The name of the application class that implements the Database- WriteCallback interface.
curam.custom.dataaccess. transactioncallback	STRING	The name of the application class that implements the Transaction- Callback interface.
curam.custom.disable.dat abase.callback	BOOLEAN	If set to true this property will disable the database callback.

Table A.10 Custom settings

A.3.7 Trace

These control what diagnostic information (in addition to errors which are always logged) is written to the application server's diagnostics file. Note that you can set the "curam.trace.*" settings independently of the "curam.trace" settings, resulting in the union of these settings.

Property Name	Type	Meaning
curam.trace	STRING	Default: trace_off. Tracing is off by default. Turn tracing on by setting the property to trace_on, trace_verbose or trace_ultra_verbose. The value trace_on is equivalent to setting curam.trace.servercalls to true. The value trace_verbose is equivalent to setting curam.trace.servercalls, curam.trace.methods and curam.trace.sql to true, while the highest trace level "trace_ultra_verbose" is equivalent to setting curam.trace.* to true
curam.trace.servercalls	BOOLEAN	Default: false. Trace server method invocations by remote clients.
curam.trace.methods	BOOLEAN	Default: false. Trace all business object (BO) method invocations.
curam.trace.method_args	BOOLEAN	Default: false. Dump arguments to BO method invocations, including the argument type. This option is only valid if curam.trace.methods is set to true or curam.trace is set to at least trace_verbose.
curam.trace.sql	BOOLEAN	Default: false. Trace SQL statements executed by entity objects.
curam.trace.sql_args	BOOLEAN	Default: false. Dump results of SQL select statements.
curam.trace.rules	BOOLEAN	Default: false. Trace Curam rules execution.
curam.trace.smtp	BOOLEAN	Default: false. Trace the calls to the SMTP server.
curam.trace.configfile.location	STRING	The location of the ".xml" configuration file which controls the output of logging within Curam.
curam.trace.oracle.cachehits	BOOLEAN	Default: false. An indicator as to whether the cache hits and misses of the Oracle prepared statement cache should be output.
curam.trace.ejb.invocatio	STRING	Comma separated list of invoca-

Property Name	Type	Meaning
n_differentiators		tion differentiator implementations.
curam.trace.suppress_optimistic_locking_detail	BOOLEAN	Default: false. Suppress SQL detail from being dumped when optimistic locking exceptions occur.
curam.trace.suppress_database_exception_detail	BOOLEAN	Default: false. Suppress SQL detail from being dumped when database exceptions occur.

Table A.11 Trace settings

A.3.8 Security

These settings configure the authentication behavior of Curam.

Property Name	Type	Meaning
curam.security.breakInThreshold	INT32	Default: 3. The number of consecutive break-in attempts that are allowed before an account is locked out.
curam.security.passwordexpiry.warningperiod	INT32	The number of days, in advance, that a user should be warned (on login) that their password is about to expire.
curam.security.loginattempts.warningperiod	INT32	Default: 1. The number of logins, in advance, that a user should be warned (on login) that they have a limited number of logins in which they must change their password.
curam.security.cache.failure.callback	STRING	Specifies the security cache failure callback class.
curam.security.disable.cache.failure.callback	BOOLEAN	If set to true this property will disable the security cache failure callback.
curam.security.identifier.minsearch.stringlength	INT32	Specifies the security Identifier Minimum Search String Length.

Table A.12 Security settings

A.3.9 SMTP

These settings configure the environment in which the SMTP client element

of Curam operates.

Property Name	Type	Meaning
curam.mail.smtp.serverhost	STRING	The default mail server that is used by Curam.
curam.mail.smtp.serverport	INT32	The port on which the default mail server is addressed.
curam.mail.smtp.connectiontimeout	INT32	The socket connection timeout value (in seconds) of the mail server.
curam.mail.smtp.timeout	INT32	The socket I/O timeout value (in seconds) of the mail server.

Table A.13 SMTP settings

A.3.10 XMLServer

These settings configure the environment in which the XML Server will be used.

Property Name	Type	Meaning
curam.xmlserver.host	STRING	The host on which the XML Print Server resides. The property may also be specified as a '/' separated list of host names in order to use multiple XML Servers. For further information please refer to the Curam XML Infrastructure Guide.
curam.xmlserver.port	STRING	The port on which the XML Print Server is listening. The property may also be specified as a '/' separated list of ports in order to use multiple XML Servers. For further information please refer to the Curam XML Infrastructure Guide.
curam.xmlserver.printer	STRING	The printer name that will be provided to the XML Server.
curam.xmlserver.tray	STRING	The printer tray that will be provided to the XML Server.
curam.xmlserver.fileencoding	STRING	The encoding that should be used for the encoding of files provided to the XMLServer.
curam.xmlserver.serialize	BOOLEAN	Specify that xml server data will

Property Name	Type	Meaning
localeneutral		be serialized in a locale-neutral way instead of being based on the locale properties on the server.

Table A.14 XMLServer settings

A.3.11 Database

These settings configure Curam for database communication.

Property Name	Type	Meaning
curam.db.readmultimax	INT32	Default: 100. Allows the developer to override the default maximum number of records returned by the readmulti (readmulti, nsreadmulti, nkreadmulti and nsmulti) operations in an application. This default value is only used if an explicit value is not set in the model.
curam.db.locktimeout	INT32	Default: 30. Allows the developer to set the lock timeout in seconds on an Oracle database when performing a singleton select FOR UPDATE. The syntax here is to append a WAIT XX clause to the statement. This default value is only used if an explicit value is not set.
curam.db.batch.limit	INT32	Default: 10. Globally defines the number of updates that can be grouped together as part of a batch update.

Table A.15 Database settings

A.3.12 KeyServer

These settings allow a customer to configure the behavior of the KeyServer.

Property Name	Type	Meaning
curam.keyserver.default.unique.set	STRING	The name of the default key set used by the application.
curam.keyserver.retry	INT32	Default: 5. The number of retries

Property Name	Type	Meaning
		that will be performed if there is a problem communicating with the key server before that problem is reported to the user.
curam.keyserver.support	BOOLEAN	Default: false. The range aware key server algorithm allows usage of group from 3 to 32,768. But as group 2 is to allocated for Cúram support.This property can be set to true to state keys generated are for Cúram support purpose.
curam.keyserver.remaining.keyblock.notification	INT64	Default: 100000000. The range aware key server algorithm supply a notification to administrators when a particular key set is nearing the end of the systems allocated range.This notification would be sent repeatedly at defined magnitude intervals before exhaustion e.g. the first message sent when there are X key blocks remaining for the key set, the next when there are X/10 key blocks remaining etc. Range Aware Key Server send these notifications only in case if there are no further ranges allocated to the system.
curam.keyserver.keyset.cachesize	INT32	Default: 1 : Specifies the number of unique ID keysets to be consumed and cached per Key Server transaction.

Table A.16 KeyServer settings

A.3.13 BatchLauncher

These settings configure the behavior of Curam when problems occur invoking batch programs.

Property Name	Type	Meaning
curam.batchlauncher.erroremail.recipient	STRING	The email address of the recipient of error emails from Curam.
curam.batchlauncher.erro	BOOLEAN	Default: false. Suppress the stack

Property Name	Type	Meaning
remail.nostacktrace		trace in the error emails.
curam.batchlauncher.default.error.code	INT32	Default: 1. The default error code returned by a batch program.
curam.batchlauncher.dbtojms.enabled	BOOLEAN	Default: false. Specifies whether deferred processing and work-flow functionality for batch programs should be enabled. When set to true the curam.batchlauncher.dbtojms.notification.host and curam.batchlauncher.dbtojms.notification.port properties must also be set.
curam.batchlauncher.dbtojms.notification.host	String	Default: localhost. Specifies whether the host on which the database-to-JMS listener is available. This property must be set when the curam.batchlauncher.dbtojms.enabled property is set to true.
curam.batchlauncher.dbtojms.contextroot	STRING	The context root used by the Curam web client. Default value = 'Curam'.
curam.batchlauncher.dbtojms.notification.port	INT32	Default: 9044. Specifies whether the port on which the database-to-JMS notification listener is available. This property must be set when the curam.batchlauncher.dbtojms.enabled property is set to true.
curam.batchlauncher.dbtojms.notification.ssl	BOOLEAN	Default: true. Specifies that the database-to-JMS notification listener on the application server is using SSL.
curam.batchlauncher.dbtojms.notification.encoding	String	Specifies the encoding of the database-to-JMS listener.
curam.batchlauncher.dbtojms.notification.batchlaunchermode	String	Specifies the db-to-jms mode for the batch launcher. 0=none, 1=once per batch launcher session, 2=once per batch job.
curam.batchlauncher.dbtojms.notification.disabled.in.standalone	BOOLEAN	Specifies that the batch launcher should not perform a db-to-jms notification when run in standalone mode.

Property Name	Type	Meaning
curam.batchlauncher.dbto jms.notification.test.stubt rigger	BOOLEAN	Default: false. For debugging batch jobs which use DBtoJMS: stubs out DBto-JMS.beginTransfer() to prevent it from creating deferred processes. JMSLiteEngine must be started to process the messages.
curam.batchlauncher.dbto jms.messagespertransacti on	INT32	Default: 512. The number of messages per transaction processed by the database-to-JMS conversion.

Table A.17 BatchLauncher settings

A.3.14 Workflow

These settings configure the properties which relate to the Workflow Environment.

Property Name	Type	Meaning
curam.workflow.disable.a udit.wdovalueshistory.bef ore.activity	BOOLEAN	When specified to true, this flag will ensure that no WDO values history audit information will be written before an activity is executed.
curam.workflow.disable.a udit.wdovalueshistory.aft er.activity	BOOLEAN	When specified to true, this flag will ensure that no WDO values history audit information will be written after an activity is executed.
curam.workflow.disable.a udit.wdovalueshistory.tra nsition.evaluation	BOOLEAN	When specified to true, this flag will ensure that no WDO values history audit information will be written before the transitions from an activity are evaluated.

Table A.18 Workflow settings

A.3.15 CTM

These settings configure the properties which relate to configuration transport manager.

Property Name	Type	Meaning
curam.ctm.landscape.nam	STRING	Default: nolandscape. The land-

Property Name	Type	Meaning
e		scape name for configuration transport manager to transport change set from source to target systems with in the configured landscape.

Table A.19 CTM settings

A.4 Application.prx - Static properties

The following properties relate to the available static properties in the Application.prx file.

A.4.1 Custom

These settings allow a developer to replace elements of the Curam infrastructure with their own customized handlers.

Property Name	Type	Meaning
curam.custom.audit.writer	STRING	Default: curam.util.internal.misc.Standard DatabaseAudit. The name of the class which will handle the generated audit information. This class must extend curam.util.audit.AuditLogInterface. curam.util.audit.DisabledAudit may be used to globally disable auditing.
curam.custom.predataaccess.hook	STRING	The name of the class that implements the interface curam.util.audit.DataAccessHook.
curam.custom.external.operation.hook	STRING	Specifies the fully qualified class name of the customized external operation Hook which implements curam.util.audit.ExternalOperationHook. An external operation is an operation callable as a remote, batch, webservice or deferred process call.

Table A.20 Custom settings

A.4.2 Security

These settings configure the authentication behavior of Curam.

Property Name	Type	Meaning
curam.security.disable.authorisation	STRING	Default: false. Suppress the authorization checks normally performed by Curam.
curam.security.casesensitive	BOOLEAN	Authentication and authorization of usernames is case sensitive by default. When this property is set to false the authentication and authorization mechanisms will ignore the case of the user. If duplicate case insensitive user names exist (e.g. caseworker, CaseWorker), authentication will fail due to an ambiguous user name. Such duplicate names will also cause the security cache to fail to initialize.
curam.custom.externalaccess.implementation	STRING	The fully qualified name of the class implementing the curam.util.security.ExternalAccessSecurity interface. This class implements the custom authentication mechanism for External Users.
curam.custom.authentication.implementation	STRING	The fully qualified name of the class implementing the curam.util.security.CustomAuthenticator interface. This class implements custom authentication verifications that will be invoked during the authentication process.
curam.custom.userscope.implementation	STRING	The fully qualified name of the class implementing the curam.util.security.UserScope interface. This class determines the type of User logging into the application, i.e. INTERNAL or EXTERNAL.

Table A.21 Security settings

A.4.3 Trace

These control what diagnostic information (in addition to errors which are always logged) is written to the application server's diagnostics file.

Property Name	Type	Meaning
curam.trace.method_hand ler	STRING	Default: curam.util.resources.Trace.Cura mMethodInvocationHandlerDe-fault. Name of a class implementing curam.util.resources.Trace.Cura mMethodInvocationHandler to perform custom method tracing.
curam.trace.dataaccess.m axstringlength	STRING	Default: 1000. Maximum length of a String or CLOB logged by the Data Access Layer when SQL tracing is enabled.

Table A.22 Trace settings

A.4.4 Environment

These settings configure the environment for your Curam application.

Property Name	Type	Meaning
curam.project.name	STRING	This parameter is required by the Rules and Workflow engines to dynamically invoke methods in the application.
curam.disable.tab.cache	BOOLEAN	Default: false. Indicates if tab caching should be disabled. Note, this only applies to caching on the server side.

Table A.23 Environment settings

A.5 Variable Property Settings

The following properties whose name is defined variably.

A.5.1 Transaction

Contains properties connected with the runtime setting of transactional options.

Property Name	Type	Meaning
\<fully qualified class-name\>.intf.\<method name\>.transaction.timeout	INT32	Used to control the transaction timeout for a single operation. The value is the number of seconds before the transaction should timeout. Format: PROJECT-NAME.CODEPACKAGE.CLASSNAME.intf.OPERATIONNAME.transaction.timeout e.g. curam.core.facade.intf.Person.createAddress.transaction.timeout=60

Table A.24 Transaction settings

A.5.2 Audit

Contains properties connected with the runtime setting of auditing options.

Property Name	Type	Meaning
curam.audit.opaudittrail	BOOLEAN	Specify whether operation level auditing for the operation 'OPERATIONNAME', within the client visible class 'CLASSNAME' of the code package 'CODEPACKAGE' is enabled or disabled. Format: curam.audit.opaudittrail.PROJECTNAME.CODEPACKAGE.CLASSNAME.OPERATIONNAME Default: determined by the option set in the model.
curam.audit.audittrail	BOOLEAN	Specify whether table level auditing for the operation 'OPERATIONNAME' of entity 'CLASSNAME' within the code package 'CODEPACKAGE' is enabled or disabled. Format: curam.audit.audittrail.PROJECTNAME.CODEPACKAGE.CLASSNAME.OPERATIONNAME Default: determined by the option set in the model.
curam.custom.external.operation.hook	STRING	Specify the name of a class which implements

Property Name	Type	Meaning
		'curam.util.audit.DataAccessHook' and which will be used to audit client-visible operation calls.
curam.custom.predataaccess.hook	STRING	Specify the name of the class which implements 'curam.util.audi.DataAccessHook' and is used to audit data access calls.
curam.custom.audit.writer	STRING	Specify the name of a class which implements curam.util.audit.AuditLogInterface and will be used to capture and write audit information.
curam.audit.audittrail.noxmlaudit	BOOLEAN	Specify whether the XML audit writer is disabled for data access operations. This saves XML from being generated for each invocation of the operation done so far. Default: false.

Table A.25 Audit settings

ℹ Note

There are two ways to turn off auditing:

• Set the 'curam.custom.predataaccess.hook' property in the Application.prx to be blank and set the 'curam.audit.audittrail.noxmlaudit' property to be true.

• Set the value of the property 'curam.custom.audit.writer' to be 'curam.util.audit.DisabledAudit'. The 'curam.util.audit.DisabledAudit' is a class that is provided by the Infrastructure that contains empty methods. Therefore the class will be called but no auditing will take place. This ensures that the 'Audit.logDataAccess' class gets called and builds up the xml that will form part of the auditing but it does not actually insert any audit records onto the database.

The first option is the preferred option.

Appendix B

Infrastructure Auditing Settings

B.1 Default table-level-audit setting

The following tables list the database operations in the *IBM® Cúram Social Program Management* infrastructure and the default value of their table-level auditing flag. This value may be overridden by setting application properties, see the *Cúram Modeling Reference Guide* for more details. Certain database operations do not support auditing, for example operations with stereotype <<ns>> which have handcrafted SQL - and these are listed with a default value of N/A

Operation Name	Default Auditing Setting
ActivityInstance.getActivityVersionDetailsByTaskID	N/A
ActivityInstance.getTaskID	False
ActivityInstance.insert	False
ActivityInstance.modify	False
ActivityInstance.read	False
ActivityInstance.readActivityInstanceByTaskID	False
ActivityInstance.readByActivityInstanceCompoundKey	N/A
ActivityInstance.readByTaskID	False
ActivityInstance.readIterationID	False
ActivityInstance.remove	False
ActivityInstance.searchByProcessInstanceID	False
ActivityInstance.searchByProcessInstanceIDAndStatus	False
ActivityInstance.setActivityInstanceStatusAndEndDate	False
ActivityInstance.setTaskID	False

Operation Name	Default Auditing Setting
ActivityOccurrence.insert	False
ActivityOccurrence.read	False
ActivityOccurrence.remove	False
AppResource.insert	False
AppResource.modify	False
AppResource.read	False
AppResource.readAllResources	False
AppResource.readByCategory	False
AppResource.readByEmptyCategory	N/A
AppResource.readByIEGScriptDefinitionID	N/A
AppResource.readByLikeName	N/A
AppResource.readByName	False
AppResource.readByNameAndLocale	N/A
AppResource.readResourceNameByID	False
AppResource.remove	False
AppResource.removeByIEGScriptDefinitionID	N/A
AppResource.removeByName	False
AppResource.removeByNameAndLocale	N/A
AuditTrail.insert	False
AuditTrail.readAll	False
AuthenticationLog.countEntries	N/A
AuthenticationLog.insert	False
AuthenticationLog.modify	True
AuthenticationLog.read	False
AuthenticationLog.readmulti	False
AuthenticationLog.remove	True
AuthorisationLog.countEntries	N/A
AuthorisationLog.insert	False
AuthorisationLog.readmulti	False
BPOMethodLibrary.insert	False
BPOMethodLibrary.modify	False
BPOMethodLibrary.read	False
BPOMethodLibrary.remove	False
BPOMethodLibrary.searchBPOMethodReferences	N/A

Operation Name	Default Auditing Setting
BPOMethodLibrary.searchByCompoundKey	False
BatchErrorCodes.getAllErrorCodes	N/A
BatchErrorCodes.insert	False
BatchErrorCodes.modify	False
BatchErrorCodes.read	False
BatchErrorCodes.remove	False
BatchGroupDesc.insert	True
BatchGroupDesc.read	False
BatchGroupDesc.readmulti	False
BatchGroupDesc.remove	True
BatchGrpGrpAssoc.insert	False
BatchGrpGrpAssoc.readmulti	False
BatchGrpGrpAssoc.readmultichildid	False
BatchGrpGrpAssoc.remove	False
BatchParamDef.read	False
BatchParamDef.readmulti	False
BatchParamDesc.insert	True
BatchParamDesc.modify	True
BatchParamDesc.read	False
BatchParamDesc.readmulti	False
BatchParamDesc.remove	True
BatchParamValue.insert	False
BatchParamValue.read	False
BatchParamValue.readmulti	False
BatchParamValue.remove	False
BatchProcDef.read	False
BatchProcDef.readAllProcesses	False
BatchProcDesc.insert	True
BatchProcDesc.modify	True
BatchProcDesc.read	False
BatchProcDesc.readAll	False
BatchProcDesc.remove	True
BatchProcGrpAssoc.insert	True
BatchProcGrpAssoc.readmulti	False

Operation Name	Default Auditing Setting
BatchProcGrpAssoc.readmultionprocessname	False
BatchProcGrpAssoc.remove	True
BatchProcRequest.insert	False
BatchProcRequest.read	False
BatchProcRequest.readallrequests	False
BatchProcRequest.readmulti	False
BatchProcRequest.readmultiuserid	False
BatchProcRequest.remove	False
BizObjAssociation.countOpenTasksByBizObjectTypeAndID	N/A
BizObjAssociation.insert	False
BizObjAssociation.modify	False
BizObjAssociation.modifyBusinessObjectID	False
BizObjAssociation.read	False
BizObjAssociation.remove	False
BizObjAssociation.searchByBizObjectTypeAndID	False
BizObjAssociation.searchByTaskID	False
CacheVersion.insert	False
CacheVersion.modify	False
CacheVersion.read	False
CodeTableData.changeTableName	False
CodeTableData.insert	True
CodeTableData.modify	False
CodeTableData.read	False
CodeTableData.removeOneCodeTable	False
CodeTableHeader.getChildCode	False
CodeTableHeader.insert	True
CodeTableHeader.joinCTHeaderCTItem	N/A
CodeTableHeader.modifyDefaultCode	False
CodeTableHeader.modifyParentCodetable	False
CodeTableHeader.modifyTableName	False
CodeTableHeader.modifyTimestamp	False
CodeTableHeader.read	False
CodeTableHeader.readChildCodeTable	False
CodeTableHeader.readDefaultCode	False

Operation Name	Default Auditing Setting
CodeTableHeader.readEntireTable	False
CodeTableHeader.readTableName	False
CodeTableHeader.remove	True
CodeTableHeader.searchByCodeTableName	N/A
CodeTableHierarchy.insert	False
CodeTableHierarchy.modify	False
CodeTableHierarchy.modifyCodetable	False
CodeTableHierarchy.read	False
CodeTableHierarchy.readAll	False
CodeTableHierarchy.readByCodetable	False
CodeTableHierarchy.remove	False
CodeTableItem.changeTableName	False
CodeTableItem.countCodeTableItems	N/A
CodeTableItem.countDescriptionSameParentCodeDifferentCode	N/A
CodeTableItem.countDescriptionSameParentCodeOnTable	N/A
CodeTableItem.countDescriptionsOnTable	N/A
CodeTableItem.countDescriptionsWithDifferentCodeOnTable	N/A
CodeTableItem.insert	True
CodeTableItem.insertWithoutTimestamp	True
CodeTableItem.listUnlinkedCodesExcludeLocale	N/A
CodeTableItem.read	False
CodeTableItem.readAllLocales	False
CodeTableItem.readAllWithoutAnnotations	False
CodeTableItem.readChildren	False
CodeTableItem.readChildrenOneLocale	False
CodeTableItem.readChildrenOneLocaleExcludeDuplicates	N/A
CodeTableItem.readDisabled	False
CodeTableItem.readEnabled	False
CodeTableItem.readOneLocale	False
CodeTableItem.readOneLocaleExcludeDuplicates	N/A
CodeTableItem.readUnlinkedCodes	False
CodeTableItem.readmulti	False
CodeTableItem.remove	True
CodeTableItem.removeOneCodeTable	False

Operation Name	Default Auditing Setting
CodeTableItem.update	True
CodeTableItem.updateWithCommentWithoutParentCode	True
CodeTableItem.updateWithoutParentCode	True
DPErrorInformation.insert	False
DPErrorInformation.read	False
DPErrorInformation.remove	False
DPProcess.insert	False
DPProcess.nkreadmulti	False
DPProcess.read	False
DPProcess.remove	False
DPProcessInstance.insert	False
DPProcessInstance.nkreadmulti	False
DPProcessInstance.read	False
DPProcessInstance.setFinishTime	False
DPTicket.insert	False
DPTicket.modify	False
DPTicket.nkreadmulti	False
DPTicket.read	False
EventClass.insert	False
EventClass.modify	False
EventClass.read	False
EventClass.readAllEventClasses	False
EventClass.remove	False
EventType.insert	False
EventType.modify	False
EventType.modifyByEventClass	N/A
EventType.read	False
EventType.remove	False
EventType.removeByEventClass	False
EventType.searchByEventClass	False
EventWait.countEventWaitsByActivityInstanceID	N/A
EventWait.countEventWaitsByEventMatchKey	N/A
EventWait.insert	False
EventWait.readByActivityInstanceID	False

Operation Name	Default Auditing Setting
EventWait.readByEventMatchKey	False
EventWait.readEventMatchDataByActivityInstanceID	False
EventWait.remove	False
EventWait.removeByActivityInstanceID	False
FailedMessage.getAllMessages	False
FailedMessage.insert	False
FailedMessage.read	False
FailedMessage.remove	False
FailedMessage.searchByMessageType	False
FailedMessage.searchByProcessInstID	False
FieldLevelSecurity.getAllOperations	N/A
FieldLevelSecurity.getAllReturnedFieldNamesByOperation	False
FieldLevelSecurity.getAllReturnedFieldsAndSidsByOperation	False
FieldLevelSecurity.getAllSecuredFields	N/A
FieldLevelSecurity.getSidForReturnedField	False
FieldLevelSecurity.getSidVersionNoForReturnedField	False
FieldLevelSecurity.insert	True
FieldLevelSecurity.setSidForReturnedField	True
FunctionIdentifier.joinFidSecurityFidSid	N/A
FunctionIdentifier.read	False
FunctionIdentifier.readAllFids	False
GroupInformation.getVersionNoForGroup	False
GroupInformation.insert	False
GroupInformation.listExcludingScript	N/A
GroupInformation.modify	False
GroupInformation.nkreadmulti	False
GroupInformation.read	False
GroupInformation.remove	False
GroupRange.insert	False
GroupRange.readAll	False
GroupRangeValid.insert	False
GroupRangeValid.readAll	False
GroupRangeValid.removeAll	False
IEGDefinitionInfo.insert	False

Operation Name	Default Auditing Setting
IEGDefinitionInfo.nsmultiGroupByType	N/A
IEGDefinitionInfo.nsmultiGroupWithoutType	N/A
IEGDefinitionInfo.nsmultiScriptByType	N/A
IEGDefinitionInfo.nsmultiScriptWithoutType	N/A
IEGDefinitionInfo.readmulti	False
IEGDefinitionInfo.remove	N/A
IEGExecutionInfo.insert	False
IEGExecutionInfo.modify	False
IEGExecutionInfo.nkreadmulti	False
IEGExecutionInfo.read	False
IEGExecutionInfo.readExec	False
IEGExecutionInfo.remove	False
IEGExecutionInfo.searchBeforeDate	N/A
Iteration.insert	False
Iteration.modifyEndDateTime	False
Iteration.read	False
Iteration.readIterationID	False
Iteration.readIterationSummary	False
Iteration.remove	False
JMSLiteMessage.insert	False
JMSLiteMessage.read	False
JMSLiteMessage.readAllByType	False
JMSLiteMessage.remove	False
JoinInstance.insert	False
JoinInstance.modify	False
JoinInstance.readByJoinMetaID	False
JoinInstance.remove	False
KeyServer.insert	False
KeyServer.modify	False
KeyServer.read	False
KeySetRange.insert	False
KeySetRange.modify	False
KeySetRange.read	False
MatchedEvtArchive.getMatchedEventsForActivityInstance	False

Operation Name	Default Auditing Setting
MatchedEvtArchive.insert	False
MatchedEvtArchive.read	False
MatchedEvtArchive.readByActivityInstanceID	False
MatchedEvtArchive.searchByActivityInstanceID	False
OpAuditTrail.insert	False
ProcEnactEvtData.insert	False
ProcEnactEvtData.modify	False
ProcEnactEvtData.read	False
ProcEnactEvtData.readByProcessStartEventID	False
ProcEnactEvtData.remove	False
ProcEnactEvtData.removeByProcessStartEventID	False
ProcEnactmentEvt.insert	False
ProcEnactmentEvt.modify	False
ProcEnactmentEvt.read	False
ProcEnactmentEvt.readAllRecords	False
ProcEnactmentEvt.readByEnabled	False
ProcEnactmentEvt.readByEvent	False
ProcEnactmentEvt.readByProcessToStart	False
ProcEnactmentEvt.remove	False
ProcInstOverflow.getWDOSnapshot	False
ProcInstOverflow.insert	False
ProcInstOverflow.removeAllRecordsForProcessInstanceWDO	False
ProcInstWDOData.getAllContextWDOForActivity	False
ProcInstWDOData.getAllWDODataForOneProcessInstance	False
ProcInstWDOData.insert	False
ProcInstWDOData.modify	False
ProcInstWDOData.read	False
ProcInstWDOData.readAllRecords	False
ProcInstWDOData.readOverflowInd	False
ProcInstWDOData.remove	False
ProcInstWDOData.removeAllContextWDOForActivity	N/A
ProcessDefinition.countDefinitionsByName	N/A
ProcessDefinition.countDefinitionsByNameAndVersion	N/A

Table B.1 Audit settings 1

Operation Name	Default Auditing Setting
ProcessDefinition.countUnreleasedDefinitionsByID	N/A
ProcessDefinition.countUnreleasedDefinitionsByName	N/A
ProcessDefinition.getHighestReleasedVersionNumber	N/A
ProcessDefinition.getHighestUnReleasedVersionNumber	N/A
ProcessDefinition.getHighestVersionNumber	N/A
ProcessDefinition.insert	False
ProcessDefinition.modify	False
ProcessDefinition.modifyByNameAndVersion	False
ProcessDefinition.read	False
ProcessDefinition.readByNameAndVersion	False
ProcessDefinition.readDefinitionByID	N/A
ProcessDefinition.readDefinitionByName	N/A
ProcessDefinition.readLatestVersionDefinitionDetailsByName	N/A
ProcessDefinition.readProcessIdentifier	False
ProcessDefinition.readProcessReleased	False
ProcessDefinition.readUnreleasedDefinitionByName	N/A
ProcessDefinition.remove	False
ProcessDefinition.removeByNameAndVersion	False
ProcessDefinition.searchAllDefinitionsSummaryDetails	N/A
ProcessDefinition.searchAllVersions	False
ProcessDefinition.searchAllVersionsByName	False
ProcessDefinition.searchByNameAndReleasedInd	False
ProcessDefinition.searchByReleasedIndicator	False
ProcessDefinition.searchDefinitions	False
ProcessDefinition.searchLatestDefinitions	N/A
ProcessDefinition.searchLatestReleasedProcesses	N/A
ProcessDefinition.searchProcesses	False
ProcessInstance.countProcessInstancesByProcessDefinitionDetails	N/A
ProcessInstance.insert	False
ProcessInstance.modify	False
ProcessInstance.modifyStatus	False
ProcessInstance.read	False
ProcessInstance.readOne	False
ProcessInstance.readStatus	False

Operation Name	Default Auditing Setting
ProcessInstance.remove	False
ProcessInstance.searchByBizObject	N/A
ProcessInstance.searchByEventWaitDetails	N/A
ProcessInstance.searchByParentProcessInstanceID	N/A
ProcessInstance.searchByProcessDetails	N/A
ProcessInstance.searchByProcessIDAndVersion	N/A
ProcessInstance.searchByTaskID	N/A
ProcessInstance.searchByTaskUser	N/A
PropDescription.countDescriptions	N/A
PropDescription.insert	True
PropDescription.modify	True
PropDescription.read	False
PropDescription.readDescriptionByID	False
PropDescription.remove	True
PropDescription.removeAllDescriptionsByPropertyID	False
Properties.countOccurrencesOfName	N/A
Properties.insert	True
Properties.modify	True
Properties.read	False
Properties.readAllByLocaleOrCategory	N/A
Properties.readName	False
Properties.readNameAndValueList	N/A
Properties.readbyName	False
Properties.readlAllPropertiesTable	False
Properties.remove	True
Properties.resetAllProperties	N/A
Reminders.clearSentRemindersByActivityInstanceID	False
Reminders.clearSentRemindersByReminderAndActivityInstanceID	False
Reminders.insertReminder	False
Reminders.scanReminders	N/A
RuleSetInformation.insert	False
RuleSetInformation.listByType	False
RuleSetInformation.modify	False
RuleSetInformation.read	False

Operation Name	Default Auditing Setting
RuleSetInformation.readDetailsWithoutDefinition	False
RuleSetInformation.remove	False
RuleSetLink.insert	False
RuleSetLink.read	False
RuleSetLink.readmultiByMasterRuleSet	False
RuleSetLink.readmultiBySubRuleSet	False
RuleSetLink.remove	False
ScriptGroupRels.dropGroupsForScript	N/A
ScriptGroupRels.insert	False
ScriptGroupRels.read	False
ScriptGroupRels.readmulti	False
ScriptGroupRels.readmultiForScript	False
ScriptInformation.insert	False
ScriptInformation.modify	False
ScriptInformation.nkreadmulti	False
ScriptInformation.read	False
ScriptInformation.remove	False
SecurityFidSid.insert	True
SecurityFidSid.joinFidSidFunctionIdentifier	N/A
SecurityFidSid.modifySid	True
SecurityFidSid.readAllFid	False
SecurityFidSid.readAllFidSid	False
SecurityFidSid.readAllSid	False
SecurityFidSid.readFid	False
SecurityFidSid.readSid	False
SecurityFidSid.remove	True
SecurityFidSid.removeSid	True
SecurityGroup.insert	True
SecurityGroup.modify	True
SecurityGroup.read	False
SecurityGroup.readAllGroups	False
SecurityGroup.readGroupsInRole	N/A
SecurityGroup.readGroupsNotInRole	N/A
SecurityGroup.remove	True

Operation Name	Default Auditing Setting
SecurityGroupSid.getFunctionSIDsForGroup	N/A
SecurityGroupSid.getNonFunctionSIDsForGroup	N/A
SecurityGroupSid.getUnlinkedFunctionSIDsForGroup	N/A
SecurityGroupSid.insert	True
SecurityGroupSid.modifyGroup	True
SecurityGroupSid.modifySid	True
SecurityGroupSid.read	False
SecurityGroupSid.remove	True
SecurityGroupSid.removeGroupName	True
SecurityGroupSid.removeSid	True
SecurityIdentifier.insert	True
SecurityIdentifier.modify	True
SecurityIdentifier.modifyNameAndDescription	True
SecurityIdentifier.read	False
SecurityIdentifier.readAllSids	False
SecurityIdentifier.readMatchSid	False
SecurityIdentifier.readSidType	False
SecurityIdentifier.readSidsInGroupSid	N/A
SecurityIdentifier.readSidsNotInGroupSid	N/A
SecurityIdentifier.remove	True
SecurityRole.getNonUsersRoles	N/A
SecurityRole.getRolesAndFunctionSIDs	N/A
SecurityRole.getRolesAndNonFunctionSIDs	N/A
SecurityRole.getUnlinkedFunctionSIDs	N/A
SecurityRole.insert	True
SecurityRole.modify	True
SecurityRole.read	False
SecurityRole.readAllRoles	False
SecurityRole.readRolesNotInGroup	N/A
SecurityRole.remove	True
SecurityRoleGroup.insert	True
SecurityRoleGroup.modifyAllOccurrencesOfARoleName	True
SecurityRoleGroup.modifyGroup	True
SecurityRoleGroup.read	False

Operation Name	Default Auditing Setting
SecurityRoleGroup.readRolesInGroup	False
SecurityRoleGroup.remove	True
SecurityRoleGroup.removeGroupName	True
SecurityRoleGroup.removeRole	True
SuspendedActivity.insert	False
SuspendedActivity.read	False
SuspendedActivity.readmulti	False
SuspendedActivity.remove	False
SuspendedActivity.removeActivitiesForProcessInstance	False
TabSession.insert	False
TabSession.modify	False
TabSession.read	False
TabSession.remove	False
Task.countAllByBizObjectAndStatus	N/A
Task.countAllByBizObjectDueDateAndStatus	N/A
Task.countAssignedByBizObjectAndStatus	N/A
Task.countAssignedByBizObjectDueDateAndStatus	N/A
Task.countByUserAndPriority	N/A
Task.countByUserAndStatus	N/A
Task.countByUserDueDateAndStatus	N/A
Task.countReservedByCategory	N/A
Task.countReservedByStatus	N/A
Task.countReservedByUsername	N/A
Task.countReservedByUsernameAndDueDate	N/A
Task.countReservedByUsernameAndPriority	N/A
Task.countReservedByUsernameAndStatus	N/A
Task.countReservedByUsernameBizObjectAndStatus	N/A
Task.countReservedByUsernameBizObjectStatusAndDueDate	N/A
Task.countTasksForReservedByUser	N/A
Task.insert	False
Task.modify	False
Task.modifyAssignedDateTime	False
Task.modifyPriority	False
Task.modifyReservedBy	False

Operation Name	Default Auditing Setting
Task.modifyRestartTime	False
Task.modifyStatus	False
Task.modifyTotalTimeWorked	False
Task.read	False
Task.readAllTasks	False
Task.readAssignedDateTime	False
Task.readReservedBy	False
Task.readStatus	False
Task.readSummaryDetails	False
Task.readTaskWithDueDate	N/A
Task.readTotalTimeWorked	False
Task.readVersionNo	False
Task.searchAllByBizObjectAndStatus	N/A
Task.searchAllByBizObjectDueDateAndStatus	N/A
Task.searchAssignedByBizObjectAndStatus	N/A
Task.searchAssignedByBizObjectDueDateAndStatus	N/A
Task.searchReservedByCategory	N/A
Task.searchReservedByDueOnDate	N/A
Task.searchReservedByPriority	N/A
Task.searchReservedByStatus	N/A
Task.searchReservedByUsername	N/A
Task.searchReservedByUsernameAndDueDate	N/A
Task.searchReservedByUsernameAndPriority	N/A
Task.searchReservedByUsernameAndStatus	N/A
Task.searchReservedByUsernameBizObjectAndStatus	N/A
Task.searchReservedByUsernameBizObjectStatusAndDueDate	N/A
Task.searchTasksByBizObject	N/A
Task.searchTasksByBizObjectAndDueDate	N/A
Task.searchTasksByBizObjectAndReservationStatus	N/A
Task.searchTasksByBizObjectUserAndStatus	N/A
Task.searchTasksByDueDate	N/A
Task.searchTasksDueInTheNextWeek	N/A
Task.searchTasksReservedDueInTheNextTimePeriod	N/A
TaskHistory.insert	False

Operation Name	Default Auditing Setting
TaskHistory.read	False
TaskHistory.search	False
TaskHistory.searchByTaskID	N/A
TaskWDOOverflow.getWDOSnapshot	False
TaskWDOOverflow.insert	False
TaskWDOOverflow.removeAllEntriesForTask	False
TransitionInstance.insert	False
TransitionInstance.modify	False
TransitionInstance.read	False
TransitionInstance.remove	False
TransitionInstance.removeByTransitionID	False
TransitionInstance.searchByProcessInstanceID	False
UserPreferenceInfo.getAllUserPrefNamesForPrefSetID	N/A
UserPreferenceInfo.getAllUserPreferences	False
UserPreferenceInfo.getAllUserPreferencesForUser	N/A
UserPreferenceInfo.getUserPreference	False
UserPreferenceInfo.insertUserPreference	False
UserPreferenceInfo.modifyUserPreference	False
UserPreferenceInfo.removeUnusedUserPreferences	N/A
UserPreferenceInfo.removeUserPreferencesForUser	False
Users.countOccurrencesOfRole	N/A
Users.modify	True
Users.modifyAllOccurrencesOfARoleName	True
Users.read	False
Users.readAllUsers	False
Users.readCaseInsensitiveUser	N/A
Users.readLocale	False
Users.readUserAndRoleNames	N/A
Users.readUsersByRole	False
Users.remove	True
WDOTemplateLibrary.countTemplatesByName	N/A
WDOTemplateLibrary.insert	False
WDOTemplateLibrary.modify	False
WDOTemplateLibrary.read	False

Operation Name	Default Auditing Setting
WDOTemplateLibrary.readAll	False
WDOTemplateLibrary.readTemplateByName	False
WDOTemplateLibrary.remove	False
WDOTemplateLibrary.searchByCategory	False
WDOValuesHistory.insert	False
WDOValuesHistory.modify	False
WDOValuesHistory.read	False
WDOValuesHistory.readByActivityInstanceIDAndExecutionPeriod	False
WDOValuesHistory.remove	False
WDOValuesHistory.searchByActivityInstanceID	False
WDOValuesHistory.searchByProcessInstanceID	False
WDOValuesHistory.searchByProcessInstanceIDAndCreationTime	N/A
WorkflowDeadline.insert	False
WorkflowDeadline.modify	False
WorkflowDeadline.modifySuspended	False
WorkflowDeadline.read	False
WorkflowDeadline.readDeadlineDetailsByActivityInstanceID	False
WorkflowDeadline.readDeadlineDetailsByTaskID	False
WorkflowDeadline.readDeadlineIDAndTimeByTaskID	False
WorkflowDeadline.readDeadlineIDByTaskID	False
WorkflowDeadline.remove	False
WorkflowDeadline.scanWorkflowDeadlines	N/A
WorkflowHistory.insert	False
WorkflowHistory.modify	False
WorkflowHistory.read	False
WorkflowHistory.readmulti	False
WorkflowHistory.remove	False
WorkflowHistory.searchByEvent	False
WorkflowHistory.searchByProcessInstanceIDAndEventTime	False
WorkflowHistory.searchByProcessInstanceIDAndUserID	False
WorkflowHistory.searchByUser	False
WorkflowHistory.searchByUserAndEvent	False
XMLArchiveDoc.insert	False

Operation Name	Default Auditing Setting
XMLArchiveDoc.read	False
XSLTemplate.insert	False
XSLTemplate.modify	False
XSLTemplate.read	False
XSLTemplate.readAllByType	False
XSLTemplate.readByIDCode	False
XSLTemplate.readByName	False
XSLTemplate.readLatestVersionAndTemplateName	False
XSLTemplate.readLatestVersionByTemplateID	False
XSLTemplate.readmulti	False
XSLTemplate.remove	False
XSLTemplateInst.deleteUsingTemplateIDAndLocale	False
XSLTemplateInst.getAllTemplateInstDetailsForTemplateIdAndLocale	False
XSLTemplateInst.getAllVersionDetails	False
XSLTemplateInst.insert	False
XSLTemplateInst.modify	False
XSLTemplateInst.read	False
XSLTemplateInst.remove	False

Table B.2 Audit settings 2

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

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.

Apache is a trademark of Apache Software Foundation.

BIRT is a registered trademark of Eclipse Foundation.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of the Open Group in the United States and other countries.

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

Red Hat Linux is a registered trademark of Red Hat, Inc. in the United States and other countries.

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